UMIACS–TR–89–84
CS–TR–2305

August, 1989

Lessons Learned in the Transition to ADA From Fortran at NASA/Goddard†

Carolyn Elizabeth Brophy
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

A case study was done at Goddard Space Flight Center, in which two dynamics satellite simulators are developed from the same requirements, one in Ada and the other in FORTRAN. The purpose of the research was to find out how well the prescriptive Ada development model worked to develop the Ada simulator. The FORTRAN simulator development, as well as past FORTRAN developments, provided a baseline for comparison. Since this was the first simulator developed here, the prescriptive Ada development model had many similarities to the usual FORTRAN development model. However, it was modified to include longer design and shorter testing phases, which is generally expected with Ada developments.

One surprising result was that the percentage of time the Ada project spent in the various development activities was very similar to the percentage of time spent in these activities when doing a FORTRAN project. Another surprising finding was the difficulty the Ada team had with unit testing as well as with integration. In retrospect, we realize that adding additional steps to the design phase, such as an abstract data type analysis, and certain guidelines to the implementation phase, such as to use primarily library units and nest sparingly, would have made development much easier. These are among the recommendations made to be incorporated in a new Ada development model next time.

† Research supported in part by NASA grant NSG–5123.

LESSONS LEARNED IN THE TRANSITION TO ADA FROM FORTRAN AT NASA/GODDARD*

by

Carolyn Elizabeth Brophy

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1989

Advisory Committee:

Dr. Victor Basili, Chairman/ Advisor
Dr. Marvin Zelkowitz
Dr. Dieter Rombach

Acknowledgements

This thesis would not have been possible without contributions from many others. First and foremost, I want to thank my advisor, Vic Basili, for all his technical support. He was also one of the overseers of this experiment. In addition, he has always been very encouraging, which is invaluable with a project of long duration.

Sally Godfrey, my co–author at Goddard for documents describing the design and implementation phases of the Ada project, has done more than any other person to help me see things more like an insider in the organization would. Without such a perspective, one cannot say much of any consequence. I also wish to thank Frank McGarry and Bob Nelson* of Goddard, without whom this study would not have taken place at all. They were the ones who first conceived of this experiment, and continued to oversee it. I consider myself very fortunate to have been assigned to work with this project.

Certainly these acknowledgements could never be complete without giving credit to the Ada team members, who spent extra time answering many questions and correcting my misconceptions. Any which still remain, are my own fault entirely. Bill Agresti, the team leader, was very helpful in orienting me in the beginning, and keeping me informed about things happening with the team. The other team members were Ed Seidewitz, Mike Stark, Dwight Shank, Bob Murphy, Betty Brinker, Pei–Shen Lo, Suzanne DeLong, Bob Schwenk, Megan Dowd, and Dave Littman.

The other members of my committee, Marv Zelkowitz and Dieter Rombach, gave me very helpful suggestions for improving this thesis. I especially appreciated Dieter's reassurance the day before my defense. Others have read this document and given me suggestions for improving it. I particularly want to thank Brad Ulery and Barbara Swain (fellow graduate students) for their help in this regard. Last, but not least, I want to thank Bill Gasarch for providing me with music tapes and sandwiches to make a number of long evenings at school more productive, and for his encouragement as well.

* He is now at NASA headquarters.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

1.1. Overview

Ada was developed in the late seventies and early eighties in order to become the standard implementation language for DoD applications. At NASA, the decision was made in 1985 to use Ada for the Space Station. With this in mind, the Flight Dynamics Division at NASA/Goddard decided to experiment with using Ada for non–critical projects to gain experience with Ada before its use on a critical project such as the Space Station[29].

The Flight Dynamics Division of NASA/Goddard develops ground control systems for satellites. Many of these satellites are now launched by the space shuttle. The division also develops telemetry simulators and dynamics satellite simulators as support projects for the ground control systems. They help train personnel, elucidate requirements for the ground control systems, and test software that will be used on board the satellites[29]. The product involved in this study is a dynamics satellite simulator for the Gamma Ray Observatory (GRO). A dynamics simulator models the control system for keeping the satellite in its proper orbit and for keeping the satellite pointed in the right direction[29,31].

A comparative case study was planned for this satellite simulator project[29]. The same dynamics satellite simulator was developed by two separate teams in parallel. When the study was first conceived, the overarching goal was to characterize and compare the Ada development process and the resulting product with the FORTRAN development process and resulting product, in order to understand, control and improve the Ada development process. The primary difference was that one team developed a simulator in FORTRAN, and the other developed one in Ada. Another difference was that the FORTRAN simulator had one subsystem which had to meet a real–time constraint[5]; the same subsystem in the Ada version did not have this constraint.

For over a decade now, the University of Maryland has done software engineering research with the Flight Dynamics Division and Computer Sciences Corporation. Together these three groups form the Software Engineering Laboratory (SEL)[43]. The systems themselves, and the development and maintenance processes studied, are those of the Flight Dynamics Division.

Past studies have led to a good understanding of the waterfall development methodology as used here[3,28]. This understanding provides a basis from which to now study the introduction of new technologies. Ada and object oriented methodologies are some of these new technologies.

1.2. Environment

This division usually develops their products in FORTRAN using a form of the waterfall development methodology[3,28]. Many of the software products are similar from mission to mission. The fact that applications are similar is important for domain expertise and for the legacy developed in this environment for code, designs, expectations and intuitions. The similarity between projects allows a high level of reuse of both design and code. Since

the applications are basically familiar ones, and since old design and code can be reused, the development methodologies which involve much iteration do not seem to be necessary.

A dynamics satellite simulator is generally about 40K to 50K source lines of code (SLOC). Concurrency has not been used much in the past. This type of application is developed on DEC VAXes. A FORTRAN 77 preprocessor, a configuration management program, and the EDT editor are used. A debugger is available, but not generally used. Software is developed by top–down, procedural decomposition and structured analysis. FORTRAN development takes place in a mature and stable environment.

1.3. Objectives of the Study

The general goal given in section 1.1 really has two parts. The first part was descriptive. There was the proposed Ada development process, and then the actual one, that is, what actually occurred. It was assumed there would be differences. Due to the newness of Ada, we were not sure what effects Ada would have on development, and how it would compare to the traditional FORTRAN development process. Characterizing the development process is intended to give a starting point from which to formulate hypotheses to use as a basis for improving development in the future. We also want to characterize the Ada products (documents and code), and compare them to the FORTRAN products.

The second part of the general goal was prescriptive. We want to find ways to control and improve the development process and products. Techniques, methods and processes which worked well with FORTRAN may not work as well with an Ada driven approach. In the end it is hoped that the processes and products used with Ada will be better than those that had evolved over years of using FORTRAN.

We learned two sorts of lessons over the course of the project: (1) affirmations that either what was done, or how it was done, works, or (2) findings that what was done, does not work well. Some findings each led to a recommendation for a new approach to some particular aspect of development or the product the next time. When this is done over a series of developments, the process and resulting products should gradually improve. This is known as the improvement paradigm[12].

The Improvement Paradigm works as follows. The characteristics of the development process and/or product, which need to be improved, need to be pinpointed. Part of doing this is to characterize the development environment itself. Once these goals are set, they need to be refined. We do this by asking questions, which can then be answered with data collected during the development process. We see by analyzing this data where our development process and product met its goals, and where it did not. We then develop hypotheses about why things went the way they did, and recommendations for improving the development process next time. When the next project starts, we go through this process again. Over time, the development process and product will gradually improve. (The improvement paradigm is also briefly described in section 2.2).

It is often difficult to determine what is actually affected by the introduction of Ada into the development process, and what is in fact due to other things. Some effects are transitional; they occur during the process of learning Ada and its accompanying technologies (e.g., object oriented design methods). One of the start–up costs is due to the fact that there is no Ada code to reuse. Reuse of FORTRAN code is important in this environment, and well established. It can sometimes be difficult to determine, which effects are transitional, and which are intrinsic, in using Ada.

Although the characteristics of the product and process, and the lessons learned come from one specific environment, many of the relationships are expected to be generally applicable to other environments. Of course the more similarities another environment has to the

Flight Dynamics Division at NASA/Goddard, the more results will apply.

A goal–driven model[1] for using metrics to fulfill our goals has been developed in the SEL[9,12,44]. Using this model, which is now referred to as G/Q/M (for goal/question/metric), we will state our overall goals for the project as:

(1) Describe the current Ada development process and the resulting product. How do these compare to the standard development process and product?

(2) Analyze the impact of the change to Ada.

(3) Provide sufficient information to develop a model for future Ada developments.

These goals generate the following questions for study:

I. Process and Product Conformance (Characterize the development methodologies, and resulting products)

(1) What was the overall process model applied during the Ada development, including the processes applied within each phase of development?

(2) What was the process applied during the standard FORTRAN development, including the processes applied within each phase of development?

(3) How well did the Ada developers understand object–oriented design, and the principles behind it?

(4) How well did the Ada developers know Ada?

(5) How well were the processes applied, which were used during the Ada and FORTRAN developments?

(6) How was the training done for Ada?

(7) How were specifications represented for Ada and FORTRAN?

(8) How well do certain design methodologies work with Ada?

(9) How was the product documented for both Ada and FORTRAN?

(10) How were implementation and testing done in FORTRAN and Ada?

(11) How did all these processes differ for FORTRAN and Ada developments? What effect did these processes have on Ada products such as documentation and code?

(12) How are all the activities and phases to be defined for Ada developments? How does this compare to the activities and phases in FORTRAN developments?

II. Domain Conformance (Application domain, and developers' knowledge of it)

(1) How well did the Ada developers know the application domain? How did this compare to the application knowledge of the FORTRAN team?

(2) What kinds of development experience do the members of the Ada and FORTRAN teams have? How does this experience compare?

III. Effect (What happened)

(1) What effect did the FORTRAN biases in the specifications have on the Ada development process and product?

[1]See Background Literature, Chapter 2, for fuller description of this model and its context.

(2) What are the effects of Ada on the Flight Dynamics development process, and the resulting product quality? How did Ada affect the following, and how does it compare to FORTRAN?
   (a) the way design was done,
   (b) the way implementation was done,
   (c) the way testing was done,
   (d) the products of each phase,
   (e) the amount of effort spent in each phase, and activities during that phase,
   (f) the amount of effort spent on each activity,
   (g) the quality of the products:
      (i) How many changes and failures were there? Were there fewer changes/failures with Ada?
      (ii) Why were the changes made?
      (iii) Where in the development process did the faults originate that eventually led to failures?
      (iv) What type of failures occurred?
      (v) How hard (costly) were changes/failures to isolate and fix?

(3) How were the FORTRAN and Ada designs different? The same?

(4) How do we compare FORTRAN and Ada products? What measures can validly compare things such as size and productivity?

(5) What effect did the various Ada features have on the resulting system?
   (a) generics
   (b) separate compilations for bodies and specifications
   (c) library units vs. nesting
   (d) tasking
   (e) exceptions
   (f) strong typing

(6) What was expected to happen (with either the development process or the resulting system)? What did happen? Why the discrepancy between expectations and reality, if there is one?

(7) Is it feasible and cost effective to use Ada (in this kind of environment)?

(8) Switching from FORTRAN to Ada means losing the benefit of experience, institutional knowledge (which is no where written down, but necessary to operations), and reuse of designs and code. Do the benefits of using Ada compensate for these losses?

IV. Feedback (What should be done next time)

(1) What kind of training is needed in order to develop systems well with Ada?

(2) If the effect of the FORTRAN biases in the specifications is negative, how should the process be changed to avoid the FORTRAN bias? Would a bias toward Ada be a good thing?

(3) How should documentation problems be dealt with? What tailoring of object oriented methodologies is required for this environment? Which design method is appropriate for the specific application, and can it be scaled up to the problem size?

(4) How should the existing development process be modified to best change from FORTRAN to Ada?
   (a) requirements analysis
   (b) design
   (c) implementation, code

(d) implementation, code reading
(e) implementation, unit testing, integration and integration testing

(5) What unexpected problems have been encountered in development? What ways have we found to deal with them?

1.4. Guide to Reading This Thesis

Chapter 1 describes the goals of the experiment, and briefly describes the environment, and how the experiment is set up.

Chapter 2 gives background for the many threads tied together in this experiment. This includes work from the SEL, background on dynamics satellite simulators, Ada, and object oriented design. Any or all of these which the reader is familiar with may be skipped.

Chapter 3 gives the research design, and rounds out the thesis partially proposed in chapter 1. The heart of the thesis is the prescriptive model for Ada development presented in section 3.2. The usual model for development used with FORTRAN is presented in section 3.1, and the questions from chapter 1 are meant to help us evaluate the prescriptive Ada model according to various criteria.

Chapter 4 is the "story" or what happened during development in chronological order. This may be used as a reference section. Chapter 5 lists things learned from the development and some recommendations, and the reader may prefer to read chapter 5, and then use chapter 4 to find the basis for those things he is most interested in. The section numbers and headings are identical in chapter 4 and chapter 5 to promote cross–referencing.

The chapter 6 format matches that of the questions posed in chapter 1. Thus, the reader whose interest is piqued by a particular question may find a brief answer here, and cross–references to where relevant information exists in this thesis.

Chapter 7 then concludes with future research.

CHAPTER 2

Background Literature

2.1. Software Engineering Laboratory

The Software Engineering Laboratory (SEL) is composed of three organizations: Flight Dynamics Division at NASA/Goddard, Computer Sciences Corporation (CSC), and the University of Maryland. It is now 13 years old, having been started in 1976. Valett et. al.[43] gives a good summary of the goals of SEL, how SEL works, and the many things that have been learned there over the years.

The purpose of SEL is to understand how software is developed at Goddard (so we have a baseline for experimental studies), and then to learn the effects of introducing various new techniques and methodologies into this particular environment. A database of information is maintained for every project done, approximately 60 projects to date. This information includes effort needed to complete various phases of these projects, and data on the changes made, and faults found.

Studies from SEL cover a wide variety of software engineering issues[10,11,27,30]. One of the directions in which progress has been made is in the field of meta-models. The development of models for making models (meta-models) began to be important when it was realized that a model developed for one organization is not portable to other organizations. Thus each organization needs to create its own models for various processes, cost estimation, etc. Guidelines for doing this are needed however[6]. The goal/question/metric paradigm and improvement paradigm, which are also meta-models, were developed in the SEL.

2.2. G/Q/M and Improvement Paradigm

In the Flight Dynamics Division, data collection is embedded in the software development life cycle. It is possible however, to collect many kinds of metrics. How does one decide what to collect? In the SEL, a paradigm now called the G/Q/M paradigm, was developed to aid in making this decision.

The goal/question/metric paradigm (G/Q/M) first appears as a guide for goal-directed data collection[9]. The steps are: (1) formulate the goals, (2) for each goal, derive questions which define the goals, and can be answered with various measures, (3) determine the metrics that will answer the questions, (4) collect the data as part of the activities of development, (5) validate the data, and (6) analyze the data. This paradigm was shown to work in a corporate environment by using it with projects in SEL after it was proposed[44]. Since formulating goals and questions is difficult, an important aspect of the G/Q/M is giving help in arriving at appropriate goals and questions[10,14].

As time went on, the G/Q/M was conceived of in a more comprehensive fashion[8,10,12]. The emphasis was on the decompositional direction (G —> Q —> M) in prior papers. Now that the paradigm is more established, the interpretation/ analysis direction (M —> Q —> G) is given more weight[8,12].

The second way the outlook has become more comprehensive is in the use of the G/Q/M paradigm inside another paradigm called the Improvement paradigm[10,12]. The purpose of the Improvement paradigm is the improvement of the software development

methodologies and/or software products in a particular environment, over the life times of multiple projects. A single project is equivalent to one cycle of the Improvement paradigm. The steps of the paradigm are: (1) characterize the environment and current practices, (2) set up the G/Q/M, (3) choose appropriate tools, techniques, and methodologies for the current project, (4) perform software development (with data collection embedded in the development process), (5) analyze the data and make recommendations for improvement, and then (6) repeat the cycle with the next project, using the feedback from the last project to revise the current software development process.

The Improvement paradigm has been used in the SEL to improve the maintenance process[36], and it has been introduced into an industrial setting as well[35]. One of the current goals is to automate the Improvement paradigm in order to use it in a given development environment[14].

2.3. Dynamics Satellite Simulators

In this case study, a dynamics satellite simulator is built during one iteration of the Improvement paradigm. Thus, the nature of such a simulator will be explained here.

A dynamics simulator must model (1) the on–board attitude control system, (2) the satellite hardware (actuators and sensors), and (3) the environment of the spacecraft[31]. The purposes for such a simulator are given in section 1.1 (Overview). The control system of the satellite keeps the satellite in orbit, and pointed in the right direction. It is typical for a FORTRAN team developing one of these simulators to have five to eight people, take 18 to 24 months, with each person averaging 1/2 to 3/4 time, and the code to have 40K – 60K SLOC[31].

The control problem simulated is a feedback cycle. The on–board computer analyzes sensor data to determine the current position and direction in which the satellite is pointing. Commands are generated for the actuators to correct any errors. Then the sensors get environmental data again, and the cycle repeats. In the simulator, the subsystem that models the environment is called the "truth model". The other subsystems in the Ada simulator have self–evident names[5].

2.4. Ada History and Other Projects in Ada

A very good summary of the history of Ada's development is given in Sammet[38]. The history of the Ada language itself, and Ada Program Support Environments (APSE) are both traced.

It is hard to find other completed projects done in Ada. It would be nice to have these, in order to be able to compare problems and benefits which occur when Ada is used for development. The largest embedded system to date is the Advanced Field Artillery Tactical Data System (AFATDS) done by Magnavox[33,45]. DoD is not the only one using Ada. The uses are beginning to be quited varied: (1) the Bank of Finland, (2) Boeing for defense system and commercial aircraft, (3) Lockheed for spacecraft control and telemetry (Milstar), and (4) the European Space Agency[33]. In addition, the FAA has decided to use Ada for its Advanced Automation System (AAS) which is the largest, most software intensive part of the new National Airspace System in development[13]. NASA has mandated Ada for use in the Space Station. The space station is a huge project with 750K SLOC of Ada estimated for support environment, and 10 million SLOC of Ada estimated for the space station itself[37]. Roy et. al.[37] cites many projects now being done at the various NASA centers in the country. 150 new Ada projects are planned in the next five years.

2.5. Nature of Ada and Ada Life Cycles

Strictly speaking, Ada is not an object oriented language, since it does not support classes and inheritance. However, it does certainly support objects, and object oriented design methodologies will clearly take full advantage of Ada.

Metrics need to be developed for Ada, where Ada is unique, that can be used in addition to older ones, to measure how well Ada is being used. Examples would be counts of packages, generics and instantiations, and some way to measure encapsulation of data types[21]. Ada is intended to support good software engineering practices during development, and thus make maintenance cheaper and easier, and lead to a reliable product.

It has generally been held that Ada would increase the length of time required for design, and decrease time required for testing and integration. Some attempts have been made to develop new life cycle models that correctly predict costs for Ada developments. Baskette[15] found that six models (Brooks model, GTE model, Softcost model, Price–S model, Cocomo) did not allow enough design time and allowed too much testing time. Kane et. al.[26] developed a model based on the cost drivers for an Ada development, which differs from the cost drivers for other developments, and explains why other models are poor predictors.

A more comprehensive discussion of the relationship between Ada developments and the life cycles Ada supports is given in Rajlich[34], though no actual projects are cited as in the two prior papers. The traditional life cycle (each phase is finished before the next one starts), the incremental life cycle (only one language for design and implementation; thus design and implementation are merged), and the semi–incremental life cycle are discussed. When these life cycles are combined with either top–down or bottom–up approaches, various paradigms for development result. Some of these are more suitable for use with Ada than others.

2.6. Object Oriented Design

Booch's book[16] on object oriented design has had great influence in the Ada world. However, this approach did not handle large projects well because it did not address object decomposition, and its method for deriving the design from the specifications in only good for small systems. (These problems are addressed in later works). Both Jalote[25] and Seidewitz[41] found this lack of decomposition a problem in Booch's methodology. Both sought extensions to Booch based on ideas from Rajlich[34] to overcome this problem. The methodology developed by Seidewitz[41] has the advantage of having been tested in a production environment.

Rajlich[34] presents two orthogonal hierarchies, important to object oriented design. One is called the seniority hierarchy, which is layers of virtual machines. The other is the parent–child hierarchy, which is the decomposition of a given package into other packages.

CHAPTER 3

Research Design

3.1. Standard Development Process

In order to understand the effect Ada has on the standard development process, we must first understand how development is generally done with FORTRAN, and the characteristics of this environment. Then we will look at how introducing Ada changed these characteristics. In the next chapter, we will see how the Ada development actually went, and the systems resulting from each development process.

Table 3.1 shows the seven phases in the standard waterfall life cycle. More than one activity type takes place in each phase, although a given phase is named for the primary activity occurring during that time. The life cycle is well understood for FORTRAN developments, and explained in detail in the "Recommended Approach to Software Development"[28]. The activities, products, methodologies and tools to use in each phase are well-defined, as well as the percentage of time each phase will take. Experienced managers also know how to estimate schedules, costs and staffing needs fairly accurately.

A dynamics simulator starts from the hardware specifications for the satellite being modeled. These are only a piece of the overall project requirements. They come from the office responsible for the total project, and are sent to a team of analysts (not the developers) who develops the specifications for the dynamics satellite simulator from them. The development stages are described below.

The first development phase is called **Requirements Analysis**. This phase begins when the development team receives the draft functional specifications. The document containing these is called a "requirements and specifications document". The development team consists of a small group of more senior developers at this stage. They identify places where the specifications are not complete, analyze the feasibility of the specifications (and algorithms) which are included, add specifications that are purely for software purposes (e.g., display and report specifications), identify all external interfaces, and identify existing code which can be reused on the current project. At the end of this phase, a meeting is held to review the correctness and completeness of the specifications. When the final draft of the specifications and requirements document, and the requirements analysis summary report is completed, a formal review known as the System Requirements Review (SRR) is held. These two reports contain the results from the activities described above. The SRR can either approve them, approve them with changes recommended, or reject them. In general, these three options are available at any review held during the development process. What occurs if the products of a particular phase have severe problems will be described at the end of this section.

The next phase is **Preliminary Design**. The team is still small. After the SRR, the subsystems are determined, and which functions (from the specifications) will go into each subsystem. If alternative designs are to be considered, it is done here. All the interfaces are completely designed down to the subsystem level, and the design of each subsystem is completed to two levels below the subsystem level. The specifications are checked to make sure they are being met. A preliminary design document is kept, and at the end of this phase is a

Requirements Analysis

IN: draft functional specifications and requirements document

OUT: feedback to requirements analysts (not developers)
who write the final functional specifications and
requirements document
requirements analysis summary report (developers)
software development plan (managers)

Team makes sure they understand requirements, note
where holes are, clarify requirements,
specify external interfaces, identify reusable
code from previous projects

MILESTONE: System Requirements Review

Preliminary design

IN: functional specifications and requirements document

OUT: preliminary design notebook
revised software development plan

Team defines system architecture, major
subsystems

MILESTONE: Preliminary Design Review

Detailed Design

IN: preliminary design

OUT: detailed design notebook
structure charts
PDL and prologs
updated software development plan
implementation plan

Team fleshes out total design; ready to code directly from it.
Data flow descriptions, all I/O, interfaces,
builds/releases are planned

MILESTONE: Critical Design Review

Implementation

IN: detailed design notebook
implementation plan

OUT: code
updated software development plan
system test plan
draft system description
draft user's guide

Team codes, code reads, unit tests, and integrates the developing
system according to the implementation plan (builds/releases)

MILESTONE: Finished code for system
Tests for each build/release passed successfully

System Testing

IN: system test plan
system code

OUT: updated system code
updated software development plan
system description
user's guide
system test results

Team does functional testing of the entire system
based on the system testing plan (which is based
on the specifications)

MILESTONE: All system tests successfully completed

Acceptance Testing

IN: acceptance test plan
system code

OUT: final version of system code
final system description
final user's guide
acceptance test results

Acceptance tests are done by developers under supervision
of an independent acceptance test team. Team tests that
requirements are met and make any changes needed.

MILESTONE: Operational Readiness Review

Standard Development Life Cycle Using FORTRAN

Table 3.1.

Preliminary Design Review (PDR).

Next is **Detailed Design**. More staff is added during this phase. After PDR, the team continues to refine the design. Prologs, program design language (PDL), all COMMON blocks and interfaces are totally finished. Structured diagrams are done of the total design, and an implementation plan is completed. This phase is culminated by a Critical Design Review (CDR).

Staff size peaks in the **Implementation** phase. Subsystems tend to be developed in parallel, according to the specifications in the implementation plan. Each sub–phase of implementation is a build. New code is created, reusable code is modified, each module is

unit tested, code read, and then put under configuration control. Unit testing is done by the same developer who developed the particular module. Code reading is done by another developer. A management plan is used to keep track of which modules (from the design) have been coded, code read, and tested. Thus management keeps track of how much of the implementation phase is still left to finish. At the end of each build, an integrated current version of the system is tested to assure it is meeting the specifications. As these things are being done, the system description and user's guide for the system are also prepared. Test plans for the system testing phase are also drawn up.

After Implementation is essentially completed, the **System Testing** phase begins. Requirement changes occur late into these projects, and therefore some implementation is often still going on. During system testing, the team does the tests specified in the system testing plan on the entire system, makes any corrections necessary, documents the results of each system test, and updates the drafts of the user's guide and system description.

When the system tests have been passed, the system then enters the **Acceptance Testing** phase. The acceptance tests are written by a totally different team based on the specifications, sometime before System Testing is finished. The acceptance testing team supervises the development team in carrying out the acceptance tests. Changes and corrections are made to the code, and the system documents are finalized. The Operational Readiness Review (ORR) is held to determine that the system is ready to go into operation.

Development is completed when the system enters **Maintenance and Operation**.

If problems come up at one of the reviews, a formal process exists to handle this. The person identifying the problem submits a description of it to the team, including the impact the problem will have if it is not fixed. The team then considers whether to make the change, make no change, or compromise with the individual submitting the report. If the person submitting the original problem report does not agree with the team's choice, a group known as the Configuration Control Board (CCB), made up of key division personnel, will determine the outcome. Such things occur rarely. When it does, however, no schedule change is made, if at all possible.

3.2. Prescriptive Development Process with Ada

We will describe here the plans for development with Ada, primarily focusing on the differences between this plan and a traditional FORTRAN development. Chapter 4 will describe what actually happened during development, and will also include quantitative data (e.g., size of product, staff hours, failure and change data).

The development of the GRO simulator in Ada was designed by the researchers to use as much of the traditional FORTRAN life cycle as possible. This gave the team a good starting point. At the same time however, they wanted to use Ada well, and plans included experimenting with the best way to develop systems with Ada in this particular environment. It was expected that some changes and unexpected decisions would have to be made at various points during the development. Future Ada projects would then modify the development process based on the things learned here.

An earlier, industrial-setting study showed the importance of training, and that it needed to include the software engineering concepts behind Ada if Ada is to be used effectively[7]. In that study, a system which had previously been done in FORTRAN was redone in Ada. Even though the members of the Ada team never saw the FORTRAN source code for this system, the Ada system design was just like a FORTRAN design. The developers still had their original biases, and training had concentrated on language only.

With this in mind, the **training** was designed so that the team could make the best use of Ada possible. Training[32] was planned to address software engineering principles,

Training

IN: *"Software Engineering with Ada", (1st edition)
*Process Abstraction Method Seminar
*Alsys videotapes
*Training exercise: Electronic mail system

OUT: *Electronic mail system
*Preliminary experience working together as group and using Ada

Requirements Analysis

IN: functional specifications and requirements document
*Composite Specification Model

OUT: *rewritten specifications and requirements document
requirements analysis summary report (developers)
software development plan (managers)

Team rewrote requirements to eliminate FORTRAN design
in the existing requirements

No Ada code to review for reuse

Preliminary design

IN: *rewritten functional specifications and requirements document

OUT: *three preliminary designs
preliminary design notebook
revised software development plan

Team defines system architecture, major subsystems for each design

MILESTONE: Preliminary Design Review

Detailed Design

IN: *preliminary design done by the chosen methodology

OUT: *detailed design notebook
updated software development plan
implementation plan

Team fleshes out total design; ready to code directly from it.
Data flow descriptions, all I/O, interfaces,
builds/releases are planned

MILESTONE: Critical Design Review

Implementation

IN: *detailed design notebook
implementation plan

OUT: *code
updated software development plan
system test plan
draft system description

draft user's guide

Team codes, code reads, unit tests, and integrates the developing system according to the implementation plan (builds/releases)

MILESTONE: Finished code for system
Tests for each build/release passed successfully

System Testing

IN: system test plan
system code

OUT: updated system code
updated software development plan
system description
user's guide
system test results

Team does functional testing of the entire system based on the system testing plan (which is based on the specifications)

MILESTONE: All system tests successfully completed

Acceptance Testing

IN: acceptance test plan
system code

OUT: final version of system code
final system description
final user's guide
acceptance test results

Acceptance tests are done by developers under supervision of an independent acceptance test team. Team tests that requirements are met and make any changes needed.

* A change from the product in the Standard Development Life Cycle.

Prescriptive Development Life Cycle Using Ada

Table 3.2.

language syntax, object oriented design methodologies, and included a team training exercise. The development team was to be trained by a graduate student from the University of Maryland. Also planned were classes on videotape (Alsys), and Grady Booch's book "*Software Engineering with Ada*" (first edition)[16]. George Cherry[19,20] (Language Auto-

mation Associates) gave his course on PAMELA.[1] The training exercise that the team would develop was to be an electronic mail system which was between five and six thousand lines of code (SLOC). The exercise was not related to the division's usual problem domain.

After training, the Ada team would be ready to start **Requirements Analysis**. However, the functional specifications and requirements document, (i.e., traditional specifications document) which goes to the development team when requirements analysis begins, actually contains preliminary design from FORTRAN dynamics simulators done in the past[22]. From the FORTRAN point of view, this is perhaps a reuse benefit, since it reuses design and therefore code.

This was considered detrimental to the experiment with Ada, since a new design suitable for Ada was desired rather than reuse of the FORTRAN designs. Therefore, because of the prior experience[7], the researchers planned for the Ada team to rewrite the specifications using the Composite Specification Model (CSM)[2], in order to eliminate these FORTRAN biases as much as possible[22]. CSM is especially oriented toward rewriting functional specifications. This is the primary kind of specifications found in this environment[2].

The other way the requirements analysis phase would differ from the usual methodology is that there was to be no search for old code to reuse. There was no old Ada code. Using FORTRAN code would have impeded the freedom to create a totally new design for Ada, experiment with design methodologies, and test the usefulness of various Ada features.

The rewritten specifications were to be part of the input into the **Preliminary Design** phase for the team, in addition to the usual requirements analysis products. One of the objectives of the design phase, in addition to designing the dynamics simulator, would be to experiment with design methodologies.

In preliminary design, the plan was to examine three design methodologies: (1) Booch's Object Oriented Design[16], (2) the Process Abstraction Method for Embedded Large Systems[19,20], and (3) the team's own methodology, General Object Oriented Design (GOOD)[39]. The optimal ones would support both this application domain, and Ada's features useful to this application domain. At the end of the preliminary design phase, one of the preliminary designs and the accompanying methodology was to be chosen to continue into **Detailed Design**.

The **Implementation** process was planned to look the same for both FORTRAN and Ada. Both were to be organized into builds and releases. The Ada development planned to use code reading, unit testing, and to do integration in the same way as it is done for FORTRAN.

After implementation, the **System Testing** process would be done similarly to a FORTRAN system test. Likewise, **Acceptance Testing** would be similar to a FORTRAN acceptance test.

3.3. Study Design

The comparative case study was conducted in the SEL. One team developed the dynamics simulator in FORTRAN in the usual manner. A second team developed the same simulator in Ada. A research group designed the case study, and observed the two development teams as the study progressed.

The two projects were designed to be as similar as possible. Both teams began development with (1) the same specifications, (2) a waterfall development methodology, and (3) worked in DEC environments.

---

[1]Booch's book and PAMELA have been updated since that time.

However, many differences existed between the projects[18,24], which prevented the Ada and FORTRAN projects from being truly parallel. These differences are summarized in Table 3.3. The FORTRAN version was the production version, thus they had scheduling pressures the Ada team did not have. On the other hand, the Ada project did not always have top priority; team members occasionally were needed first on other production projects. This was also the first time any of these team members had done an Ada project, while the FORTRAN team was quite experienced with the use of FORTRAN. The standard development methodology was modified for use with Ada, based upon prior experience, and assumptions about how an Ada development should be different from a FORTRAN development (See section 3.2). The Ada team required training in the language and associated development methodologies, while the FORTRAN team did things in the usual way[28]. The Ada team also experimented with various design methodologies; this was necessary to find which ones would work better for this development environment. The Ada development environment was in a state of flux, unlike the very stable FORTRAN environment, due to experimentation with the design methodology and Ada related questions which arose throughout the development. In switching to Ada, the legacy of reuse for design, code, intuitions and experience are gone, and will be rebuilt slowly with the new language.

The philosophies of development were different between the two projects. The Ada team consistently applied the ideas of data abstraction, information hiding, and the state machine concept to their design development. The FORTRAN development used structural decomposition and procedural abstraction.

Both the FORTRAN and Ada teams started in January, 1985. The Ada team began with training in Ada, while the FORTRAN team began immediately with requirements analysis. The FORTRAN team delivered its system after completing acceptance testing in June, 1987. The Ada team finished system testing in June, 1988. Complete acceptance testing was never carried out on the Ada system.

The two development teams were similar in size. During design, each team had seven people. The maximum number on the FORTRAN team was ten, and on the Ada team was eleven. The teams reached their maximum during implementation. The nature of each team was different, however[22]. The Ada team had more overall experience in development and with more languages, but the FORTRAN team had more experience with simulators, per se.

Ada	FORTRAN
Experimental version	Production version
Schedule, but no pressure	Schedule pressure
New methodologies	Usual methodologies
Object Oriented Design	Structured analysis
Built from scratch	Reused design, code, experience
Fluctuating environment and underdeveloped process for development	Stable environment and stable development process

Table 3.3. Project Comparisons.

3.4. Data Collection

Data was collected from four sources[23,24]. **Standardized forms**[2] collect information which is of general interest across all projects in the SEL. These forms, with occasional updates, have been used for many years in SEL with the FORTRAN projects, to collect information during every development phase. This information provides a comparison of the current Ada and FORTRAN projects to each other, and to past FORTRAN projects. Table 3.4 shows the type of data collected.

Interviews with the Ada team provided qualitative information about the project which would have otherwise been lost. At the end of detailed design, and again at the end of implementation, each team member was asked many open-ended questions about the phase which was ending. Team members were also asked questions at other times in order to either clarify issues for the observers, or to understand team views of current development issues. Ada team members as well as FORTRAN team members answered questions regarding standard FORTRAN development practices.

Observers went to team meetings during the design and implementation phases. This allowed first-hand observation of some of the problems each developer had, and the suggestions others had for solving each problem.

Static analysis of the code itself includes a growth history of the code, that is, size and number of modules at various stages of development.

3.5. Lessons Learned Organization

A "lesson learned", for purposes of this presentation, is considered to be a fact established through empirical observation. It may even be a recommendation, but not necessarily. Specifically, it is a piece of information which answers questions established through the G/Q/M.

Estimation of effort	To set up schedules, and assign staff
Actual effort	To determine cost; Develop models for making estimates on future projects
Changes	Types of changes occurring and when; Cost and quality comparisons across projects
Failures	Types of faults occurring, when in development did problem originate; Cost and quality comparisons across projects
Time to isolate and fix changes/failures	Costs to project

Table 3.4. Data Collected from Forms.

[2]See Appendix: Data Collection Forms.

Presenting the lessons is a problem on two levels. First is the problem of organizing all the lessons themselves; then follows the problem of how to present each individual lesson. We have discovered that this issue can be a research topic all by itself. There are several possible ways to organize the lessons from this case study. Some of these ways overlap with each other.

The first general way to organize is by subject. Various subject organizations exist: (1) by life cycle phase (i.e., chronologically), (2) by Ada feature, or (3) by the software engineering concept involved. In the latter case, we could use categories like reuse, information hiding, maintenance, methodologies, and changes/faults.

Another type of organization is according to where the lesson falls on some type of linear scale. (1) One example of this type is to categorize by the importance/risk of the lesson to the project. Is it essential to project success? Just helpful to success? Or is it nice to have (e.g., might lower cost), but the project is still a success without it? (2) Another category is by specificity of the lesson learned. These would be (a) specific to flight dynamics application, (b) specific to GRO product or process, (c) only a first Ada project effect, (d) specific to Ada in any environment, (e) related to the waterfall development process, (f) related to Ada use when FORTRAN was the prior language used, and (g) language independent software engineering lessons.

Each of these classification schemes for the lessons learned, along with the lessons themselves, have their benefits in achieving the goals stated earlier. They help us to understand and characterize the current environment and process more deeply. They also help us to understand the problems of introducing the new technologies which Ada encompasses.

The classification scheme that will be used here is chronological. Chapter 4 will give the data in chronological order. Chapter 5 then gives the lessons in the same chronological order; even the section headings are identical for both chapters. This is meant to make cross–referencing lessons and the supporting evidence simple. Each lesson is derived from one or more of the data sources discussed in section 3.4 (Data collection). When presenting a given lesson, the lesson will be listed, and cross–references will be listed, if helpful. Chapter 6 lists the questions posed in chapter 1 again, along with the sections of this document where the answers can be found, and brief answers.

CHAPTER 4

Observations

4.1. Introduction

All the data we have collected during the project is reported in this chapter. The design and particularly the implementation phases are emphasized; most of what we learned comes from this part of development. The detailed design and implementation phases are the ones we had the opportunity to observe. Others have written reports on the other phases[32, 42].

Institutional intuitions, development methodologies and designs, code, and other products of development are either partially or totally lost during the transition from using FORTRAN to using Ada. Loss of this legacy is a big part of the initial cost of switching development technologies. Some of the legacy is process expectations (e.g., phase definitions, how to do a design, how to code read, how to do unit testing, or integration). Some of the legacy is product expectations (e.g., what a design document looks like). The key question is, can this reuse legacy actually be improved upon by use of Ada? That is part of what we hope to gain with the transition.

Many new questions arose when the team tried to decide how to map the current state of the design into products that were as closely related as possible to the products expected from a FORTRAN development. That is, they still attempted to follow the old FORTRAN guidelines for phase definitions.

Sometimes the Ada team faced questions which would never arise in a FORTRAN development (e.g., use of Ada features, or recompilation issues). At other times, they found that the answers to the questions which could arise for either development language should be answered differently for Ada (e.g., documentation, or how to view the various parts of the Ada development life cycle).

4.2. Effort and Size Estimates

FORTRAN estimates were made, based on past experience with similar projects. It was then assumed that the Ada project would take about three times as much effort, due to the newness of the technology. The total number of man–months estimated for the Ada project was 175; the total number of man–months estimated for the FORTRAN project was 58. No prior experience existed with Ada, although the common expectation was that the design phase would be longer, and the test phases shorter. Accordingly, the Ada project planned on more time in design, especially since they were making a brand new one rather than reusing an old design. Less time was planned for implementation, integration, and testing. Table 4.1 gives an overview of the estimates made at the beginning of the project.

The Ada project actually took about 23,000 manhours, and the FORTRAN project actually took approximately 15,000. In reality, nearly every Ada phase turned out to be longer than for the FORTRAN project. (See Phases — Overall, section 4.9 and subsections, for actual effort data). However, the Ada development effort overall was still less than estimated; the FORTRAN development effort was more. The most important reason for the difference in amount of time required for each project was that the FORTRAN project was

truly a production project, while the Ada project was experimental. This allowed the Ada team the luxury of making enhancements in their version not required by the specifications, which increased time spent on the project. Other experimental aspects (e.g., doing preliminary design with three different methodologies) also make the Ada project take longer than the FORTRAN project. In addition, since this was the first Ada project, learning added additional time in every phase.

Calendar time is different from manmonths in that no individuals were full time on either project. The Ada project was initially planned for 24 months calendar time. The FORTRAN project was initially intended to take 16 months calendar time[31]. At some point, the FORTRAN estimate was revised up to 21 months, and the Ada project estimate changed to 23 months. (See Table 4.2). The Ada project was designed to have some lead time for training. Calendar time for the Ada project actually took much longer than planned, due in part to the fact that it was experimental. Since almost every developer was also part of some other production project, the production projects tended to get priority.

Before the projects began, the final size of the Ada and FORTRAN systems were expected to be about the same. Table 4.3 shows how expectations regarding size changed during development. Source Lines of Code (SLOC) are defined as the number of carriage returns. During design the size of the finished Ada system was estimated to be 90K; during coding, the Ada system actually reached this size. These counts and estimates include comments. Actual size of the FORTRAN system, excluding blank lines and comments, is 25.6K SLOC. The actual size of the Ada system, excluding blank lines and comments, is 59.1 SLOC. The number of executable statements in each system is approximately the same; 22,840 Ada statements, and 22,300 FORTRAN statements[24]. The size of each system in SLOC, counting every line of any type, is shown in Table 4.3.

4.3. Training

The training lasted for about six months, and was equivalent to about two months full time for each person[22]. Section 4.9.3, Time spent in each major activity, compares the actual time (manhours) spent by both the Ada and FORTRAN teams in each activity. Section 4.9.4, Effort by phase, looks at actual time each team spent in each phase. A description of the training is given in section 3.2, where the Ada development plans are given. An experienced Ada person was available to the team during development. This person was a consultant, not a team member. Training experiences and lessons learned are discussed in[32].

4.4. Requirements Analysis

The specifications were re–written using the Composite Specification Model (CSM)[2], in order to eliminate the FORTRAN preliminary design from the specifications. CSM allows a system to be represented by multiple views. The first benefit derived by re–writing the specifications was that the team had a better understanding of the problem they were to solve. This understanding was deeper than it would have been from only analyzing the specifications. This is especially important since the Ada team had less experience with dynamics simulators overall than the FORTRAN team had. During design the team felt that the re–writing project had also prevented them from putting off important questions that should be handled in design until implementation, when major design changes could have been required. We shall see later that some important issues were missed anyway. (See especially section 4.7.1.6, Concurrency and tasking, and section 4.7.1.10, Strong typing).

The specifications which resulted from applying the CSM were considered to be entirely language neutral by the team, though some design was still there, nonetheless. But there was less preliminary design than in the original specifications. There did appear to be some

bias toward an object oriented design methodology, though no one considered this kind of bias to be a drawback. The new specifications allowed the team freedom to experiment with three different design methodologies The benefits and drawbacks of each methodology was to be explored.

4.5. Preliminary Design

The team was large from the outset of the project (seven developers from the time training began). Usually a team is not built up to this size until implementation. This was good for training, and for experimentation in preliminary design. It may be a bit large for efficiency in the early phases of non–experimental projects.

Preliminary designs were done with three methodologies: Booch's Object Oriented design methodology, Cherry's Process Abstraction Method for Embedded Large Applications (PAMELA), and General Object Oriented Design (GOOD)[17,22]. Booch's methodology[16] (first edition) accepts specifications written in ordinary English. Booch's notation has the advantage of being clear and describing objects well. It shows which objects use which other objects. Its major disadvantage is that the methodology as it existed during the time the team did their design, could not handle large projects well. Hierarchical structure could not really be represented, and no way existed to represent data flows. The technique for deriving the design from the specifications is inadequate for large specification documents[22].

Another preliminary design was done with the Process Abstraction Method (PAMELA)[19,20]. Since this object oriented methodology was developed for use particularly with embedded applications, it is no surprise that it is oriented toward tasks. Since it also is designed to handle large applications, it can represent hierarchical designs. It also shows data flows and some control flow. PAMELA's disadvantage was that it did not deal well with the decomposition of objects that are ultimately handled with sequential code.[22,41]. This application had many more sequential parts to it than parallel parts, and that is typical of applications in this environment. Like Booch's methodology, PAMELA has also been modified since the time the team used it on this project.

During experimentation with these various methodologies, the team began development of their own version of object oriented design, which they named *General Object Oriented Development*, or *GOOD*[39,41]. The methodology evolved concurrently with the development of the preliminary and detailed designs. The team felt this methodology combined the positive aspects of the other two object oriented methodologies without the disadvantages.

The design notation uses *object diagrams* which pictorially describe control flow, and show where any given object fits into the two orthogonal hierarchies which the methodology uses. (See Figure 4.2, in section 4.6.1, Comparison of the Ada and FORTRAN designs, for an example of an object diagram). Here the influence of Rajlich[34] also shows. One hierarchy is parent–child, which is a decompositional hierarchy. The other one is a seniority or abstraction hierarchy, similar to levels of virtual machines built one on top of the other.

Object descriptions are text–like data flow descriptions for the design. Originally, data flows were represented in the object diagrams. Preliminary design documents with GOOD were done this way. Object descriptions had evolved to replace the data flows in the object diagrams by the time the team was in detailed design. This made the design's representations much clearer[41].

The objects were considered as state machines. Each part of the design notation easily translated into Ada. Objects become packages, procedures are procedures or functions, states are variables and data structures, actors become tasks, and control flow arrows (called "communications") become procedure/function calls or entries into tasks[39]. The new documentation developed to accompany this methodology, object diagrams and object

descriptions, replace the structure charts usually used in FORTRAN developments. The principles this methodology is built on are two that are generally recognized as important in software engineering for creating systems of high quality: abstraction and information hiding.

4.6. Detailed Design

Structural decomposition, which is the design methodology usually used for these applications when they are done in FORTRAN, was not considered suitable for Ada development or experimentation, because it did not encourage use of many Ada features. It only includes a subset of the ideas inherent in the object oriented methodologies; Ada was intended to handle the fuller set.

One methodology, of the three used in preliminary design, was chosen to use for the rest of design. GOOD was the object oriented methodology chosen to continue with during detailed design. GOOD is a synthesis of ideas from both Booch and Cherry's methodologies along with additional adaptations in order to suit the methodology to Goddard's environment. It was the OOD methodology best adapted for a large, complex, primarily sequential application.

Since the design phase of this project ended, GOOD has been expanded to be more than a methodology for design only. It is intended to apply to the whole life cycle. There were some shortcomings with the methodology however. Concurrency could not be represented well, even though some thought had been given to this[39]. (See section 4.6.2, Ada design documentation, and section 4.7.1.3, Effect of design on implementation).

While the team knew the importance of reuse, it was not a high priority in this particular project to design with future reuse in mind.

4.6.1. Comparison of the Ada and FORTRAN Designs

Many similarities exist between the two designs, since both systems model the same control problem. But the resulting designs also have many important differences, due to the different design methodologies used to produce them. Seidewitz[41] discusses the two designs; Agresti et. al.[5] give a very complete discussion of the issue.

Figure 4.1 and Figure 4.2 illustrate the FORTRAN[41] and Ada[4,5,41] designs at a very high level. The FORTRAN design is reused from past simulators. The Ada design, at the highest level, Figure 4.2(a), consists of two objects that run concurrently. Figure 4.2(b) illustrates the level just below this in the parent–child hierarchy for the "GRO Simulator" object. That is, when "GRO Simulator" is decomposed, we get the other major subsystems in the Ada design. These are examples of the object diagrams used with Ada. The objects higher in a given diagram are senior to objects lower in the same diagram. This is the manner in which the seniority hierarchy is represented in object diagrams. A senior object can use the services of a junior object, but the reverse is never true.

The Ada system is one program, while the FORTRAN system consists of three programs; the Postprocessor subsystem and Profile subsystem are each separate. Both systems consist of a total of five subsystems. Thus, the central FORTRAN program contains the TM, OBC, and SCIO (Simulation Control and I/O). Figure 4.3[5] shows how the functions which must be performed are distributed among the subsystems in the FORTRAN and Ada systems. The On–Board Computers (OBC – there is a primary and backup computer) control the satellite orbit, and the direction in which the satellite points. The OBC contains the same functions in both systems. The Truth model models the environment, sensors and actuators. The Truth Model (TM) for the FORTRAN system had to meet a real time constraint, which is part of the reason for the design difference here. The use of the Profile

program reduces some of the calculations done by the TM in the FORTRAN system. The Ada TM was designed to more closely mirror reality; it does so however with a performance cost. In addition, the TM is junior to the OBC in the Ada system, which makes it passive. In the FORTRAN system, the OBC and TM are of equal status[5,41].

Agresti et. al.[5] have analyzed the data flows in each system. Since the FORTRAN system is composed of three programs, there are more external data flows in this system than in the Ada system. The FORTRAN internal communications all use COMMON blocks. The Ada system has more internal data flows than the FORTRAN system does. However, the total number of data connections (internal and external) between subsystems is greater in the Ada system. But, since variant records are often used to bundle this data, the Ada system has fewer data items using these connections[5]. The Ground Command Database and Parameter Database are global, encapsulated data stores for the Ada system[41].

The final difference between the two systems is in the timing of the TM and OBC. The control loop itself is similar in both systems. However, the TM and OBC timing are independent of each other in the Ada system and not in the FORTRAN system.

4.6.2. Ada Design Documentation

The design documentation naturally developed during the course of the project, alongside the design methodology. Though unavoidable due to the nature of this project, this constant change hindered the development of the design[17,22]. However, the methodology development was part of the learning phase involved in an initial Ada project, and part of its experimental aspect.

Keeping the design consistent is a lot of work with these detailed object diagrams and object descriptions. This is even more of a problem, with the evolution of the notation that was occurring during this project. Communication between team members is made more difficult, since they need to adapt to the changing notation at the same rate the changes keep occurring. This also hinders understanding between team members and between the team and management.

The developers found that object diagrams worked very well as a means for representing the design for the sequential parts of the system. However, they were not adequate without some revision, for use with tasks. (See section 4.7.1.3, Effect of design on implementation).

Changing from a well-known type of documentation (structure charts) to a new type of documentation (object diagrams) created problems. Manager and developer understandings and intuitions developed in the old environment no longer apply. Yet they used these old understandings. However, since they are not explicit, the managers and developers do not easily recognize that they are doing this. So when managers and developers who are unfamiliar with the new type of documentation try to apply their old intuitions, miscommunication results. These miscommunications may or may not be recognized.

One indication of this loss of understanding was revealed during reviews (PDR AND CDR). Less precision in structure charts and reviews was acceptable for FORTRAN developments than was acceptable with the Ada development, because managers knew what to expect. The Ada team presented the same types of information in the new documentation as the FORTRAN team did. But managers had much more difficulty understanding the object diagrams, and they interpreted them as if they were FORTRAN structure charts.

The design notation did not adequately show the strong degree of coupling existing between some of the objects. (See also section 4.7.3.1, Factors complicating unit testing and integration). Modifications to the design representation and methodology during the duration of this project improved this for the sake of subsequent projects.

4.6.3. Timing of Reviews and Phase Boundaries

The usual guidelines for when to have a PDR or CDR applied during a FORTRAN development, appeared very arbitrary with an Ada development. The phase boundaries seemed much less clear in the Ada development project than in the FORTRAN project. The FORTRAN guidelines were actually arbitrary in the FORTRAN context also, but due to their familiarity, it did not seem so.

Requirements for PDR/CDR are what make the cut–offs between these phases for FORTRAN developments. The requirements are described (in part) in terms of the documentation usually used with FORTRAN. For Ada the products are different, so there's no feel for where to draw the line. Since the points are arbitrary even with FORTRAN, and the methodologies are so different, there is no good way to convert the FORTRAN cut–offs to Ada cut–offs. If the FORTRAN points were less dependent on the FORTRAN product representations (e.g., structure charts) and the FORTRAN specific aspects of the process, this translation may not have been so difficult.

In addition, phase boundaries between requirements analysis and preliminary design seem blurred when specifications contain some of the design. Boundaries between preliminary and detailed design are blurred since detailed design is just further refinement of the preliminary design. The boundary between detailed design and coding is blurred by the ability to represent the design with Ada specifications, which are compilable, and to have compilable PDL. In other words, phase boundaries tend to be thought of as indicating an abrupt change in the primary activity being performed in development. This is not the case anyway, and even less so with an Ada development.

The team felt pressured to go ahead with CDR a bit sooner than they would have liked. Some team members felt prepared for CDR, since they knew the design of the system so well due to the rewriting of the specifications. Others felt unprepared due to the newness of the methodologies, representations, and new questions that were always arising with Ada. Time spent on the project peaked markedly just before CDR. How should the state of the design be mapped into the usual things expected at PDR and CDR? The team had wanted to use compilable PDL for CDR, but ran out of time.

The team felt the design really was not complete at CDR. Thus the first few months of the implementation phase were considered a continuation of the design phase. Specifications for the system were also entered at that time, as well as utilities required (See section 4.7.1.1, Builds).

4.7. Implementation

On the surface, the implementation process looked the same for both FORTRAN and Ada. Implementation was top–down. Implementation plans for both systems were organized into builds and releases. Both used pseudocode PDL and prologs for documenting each module which described the purpose and I/O for the unit.

The nature of the builds were different, however. The first build for the Ada project was coding the specifications and the utilities. Such a build would not exist in FORTRAN. Some of the utilities were math functions usually provided by a FORTRAN library, but not available in Ada. Others were application specific utilities. Early definition of the specifications, and the possibility of compilable PDL, means the interfaces must be defined early also.

The team originally had four builds planned at CDR time. During implementation the last two were collapsed into one. The development of release 1 and release 2 (There were two releases, total) were then done concurrently in separate libraries. (See section 4.7.1.13,

Library structure). Coding and unit testing were carried out for the second release, while integration and integration tests were done for the first release.

The size of the team was increased from seven to eleven persons during implementation. The maximum size of the FORTRAN team was ten.

4.7.1. Coding

4.7.1.1. Builds

Build 0 consisted of the compilable specifications for the whole system plus utilities. This Ada project did not generate any compilable specifications until implementation due to pressure to get to CDR. Both general and application specific routines were grouped together into one large utilities package. Build 1 consisted primarily of the User Interface. At the highest level in the system, the User Interface and the Simulator are the two objects running concurrently. Builds 2 and 3 are primarily for constructing the Simulator part of the system.

The User Interface was the most difficult part of the system to code. All the tasks (eight or nine) are here except one. The User Interface also uses many modules from the Simulator subsystem, which was not yet built; thus many stubs were required for testing.

4.7.1.2. Coding Issues and Standards

Coding and style standards were set at the beginning of the project. For the most part, the Ada style guide[40], which contained these standards was helpful. The style guide took some getting used to since the style was different than that used in Ada training books.

Several coding issues arose due to the newness of Ada in this FORTRAN environment. Some seasoned FORTRAN programmers who were added to the project in the implementation phase were uncomfortable with the information hiding concept. There was a distrust of what they could not "see", and FORTRAN is a much more transparent language than Ada.

The team learned better ways to code things stylistically. For example, one team member found that he should have used functions in declarations to calculate values used only as constants, rather than declaring procedures to do this.

4.7.1.3. Effect of Design on Implementation

In one important way implementation was promoted by the design. Most team members found it easy to code from the design documents. It is interesting to note that the developers who felt this transition was the easiest were the same developers who had been on the project from the start. Developers added during implementation who had not been trained in OOD methodologies (though they had read about them) did not necessarily find code writing from the design documents so easy. Some had been on an Ada project before.

The correspondence between objects and packages, or actors and tasks was very straightforward[39]. However, the one type of unit difficult to code from the design was tasks. This is due to the nature of tasks. They require the ability to express more in the representations, which was lacking. The representation for the design worked well for sequential code, and showed dependencies, but lacked the ability to represent control interactions, which is required by tasks. (See section 4.6.2, Ada design documentation).

4.7.1.4. Design Additions and Changes

During implementation, more redesign was done with the Ada project than it is possible to do with a production simulator, due to time constraints. As proficiency with Ada

grew, some of the modules done early were redone and improved, such as the Report Generator and Simulation Results. In these modules, the new versions took advantage of Ada's overloading feature.

With FORTRAN projects, there are generally some design additions. In early implementation, additions to the design are more common than design changes. The Ada project was no exception. One example of such an addition to the Ada project was the Debug Collector. Normally these are built into FORTRAN projects, even though they are not required.

Later, small design changes are common, to fix parts of the system that do not work due to design problems. A bad design decision, where the system still works efficiently enough, is allowed to stand.

The additions and design changes (they are both at the same time), which had the greatest impact on the Ada project, were the tasks in the User Interface. (See section 4.7.1.6, Concurrency and tasking). Such a construct, by its very nature, would have more impact than any constructs available in FORTRAN. The team's opinion was that the design was not substantially changed during implementation.

4.7.1.5. Library Units vs. Nesting

The Ada dynamics simulator had library units at the top levels, then it was nested sometimes eight to ten levels deep[24]. A library unit is the outer lexical level of a piece of code. Nested units are inner lexical levels.

Systems have different properties, in part determined by the number of nested units versus the number of library units used. One such property is the automatic enforcement of information hiding due to the program's static structure. With library units, visibility in the program structure is explicitly created by using "with" statements.

A disadvantage of nesting is increased recompilation. To decrease recompilation with library units, the library units must be "with'd in" at the lowest level possible in the system, and only when the context provided by the particular library unit is required. Then the pieces of the system dependent on other other pieces of the system are smaller than is the case with nesting. Dependencies are assumed between sibling units when nesting is used, but must be explicitly stated when library units are used.

Another nesting disadvantage was that it was harder to read the code and to trace problems back through nested levels than through library unit levels. The "with" clause and dot notation for naming tells you where the source of a piece of code is. For this reason, making changes is easier with library units than with nesting for larger systems.

Reuse is not encouraged by nesting. Some context may be required, which can be brought along with library units, but extra, unnecessary code is not included. The Ada project done after this one spent quite a bit of time unnesting code in order to reuse it.

Unit testing revealed another nesting disadvantage. In order to "see" inside nested units during unit testing from the test driver, the debugger was required. (Also see section 4.7.3.2, Debugger). This would not necessarily be so with library units.

The only real disadvantage found for library units was a more complex library structure. With nesting, the complexity is put inside the program pieces rather than revealed at the library level. (Also see section 4.7.4.4, Library units vs. nesting [during integration]).

4.7.1.6. Concurrency and Tasking

Concurrency was new to many of the team members, and there were many misunderstandings among them as to how tasking worked. The original design called for only two

tasks. One was in the User Interface subsystem, and one in the Simulator subsystem. Thus each of these would run asynchronously. In order to control problems which arose during implementation, eventually six or seven more tasks were added to the User Interface portion of the system.

Lateral calls to sibling tasks were put into the design, as tasks were added to the system. It did not seem that the developers realized at first that cycles were potentially present, and therefore deadlock could occur. The cycle was broken by creating a parent task to call any of the siblings. That is, a controller task was created to control information flow between its children tasks.

Perhaps the presence of cycles was one of the consequences of the "local view" taken of tasking during design in order to preserve information hiding. No high level overview of the interaction of tasks was done; instead they were viewed locally only. This may have been due to going a bit overboard with the philosophy of information hiding during design. Since designers would usually be more experienced than the implementors, it is beneficial for a project to have the control interactions and overall task interactions worked out during design by the more experienced personnel.

Tasks were also added by individual developers. Some functions, which looked like good candidates for tasking from a developer's local point of view, were made into tasks. The whole team agreed to the change, but without doing a global analysis. If a global analysis had been done, the conclusion would have been different. Unawareness of a particular task's function at the global level meant some tasks could not really operate concurrently as planned, but had to wait for other program parts to finish before they could operate.

4.7.1.7. Generics/ Separate Compilations

Some Ada features were easier to code than others. Generics were fairly easy to implement and they reduce the amount of code required.

Separation of specifications and bodies for compilation is quite beneficial and also easy to implement.

4.7.1.8. Interface Development

The specifications give an early, high level view of the system. The interfaces (directly supported by the specification construct and strong typing in Ada) have to be defined early. This could have the benefit of testing the validity of the design early in development. By the end of design, the team felt the interfaces developed were one of the more successful aspects of the design process. They were certainly easier to design than they usually are in FORTRAN.

The team had problems however, because they had to keep changing the interfaces. It is usual for requirements to continue to change throughout development, so it is hard to have an accurate, high level view of the interfaces early in development. In addition, many of the changes were due to type changes; the team had to define things sooner than they were ready to, with the strong typing of Ada. When the type changed was a global one, many interfaces would be affected. Other changes were due to functionality changes or parameter changes in particular unit(s).

4.7.1.9. Global Types

The project used a global types package that was "with'd in" everywhere. This is typical for a FORTRAN dynamics simulator also, and continuing this practice seemed to be a good idea. At first this was thought to be an advantage; well into implementation however, the team had found otherwise. One problem is recompilation. If the types change frequently

due to often changing requirements or team inexperience, the whole system will have to be recompiled each time the global types package changes.

4.7.1.10. Strong Typing

Strong typing was one Ada feature which made coding more difficult. Strong typing is hard to get used to when used to weakly typed languages such as FORTRAN. While the team had experience with many languages, some were veteran FORTRAN programmers, and most of their production experience is FORTRAN.

Besides interface problems (see section 4.7.1.8, Interface development), other problems resulting from strong typing were increased code size and an unwieldy number of types to handle during coding. The tendency existed to create too many new types. During design, a brand new type would be created with a strict range, appropriate for one part of the application. Then another area of the application would need a similar type. A subtype could have been used, if the original type had been more general, but the range on the original type was found to be too restrictive. So a whole new type would be created, including a whole new set of operations. Problems with types began to appear toward the end of design; however, the extent of the problem was not fully realized until deep into implementation.

Despite all this, strong typing did have its helpful aspects, too. The compiler found many mistakes early that are usually not found until execution.

4.7.1.11. PDL and Prologs

FORTRAN and Ada both used pseudocode PDL (program design language) and prologs for documenting each module, which describe the purpose and I/O for the unit. No algorithms were included in the prologs for most Ada units. This is generally included for FORTRAN. Purposes of the unit, other units used, and other units called were included in the Ada prologs. The function was designed at the package level during the design phase, rather than at the procedure level as is usually done with FORTRAN.

4.7.1.12. Meetings

The Ada team required many more meetings during implementation than the FORTRAN team required for several reasons. Since Ada was new, they shared things they learned with each other. The team also felt they had a poorer understanding of the functions of procedures being used than they usually did with FORTRAN developments. The functions were only described down to the package level in the design, rather than down to the procedure level. They had to take extra time to discuss these functions. Sometimes everyone assumed that conversions or initializations were done by someone else's procedure, and coded their own units accordingly. Meetings were also required during unit testing and integration to prevent inconveniences with unexpected recompilations, since recompilations were very slow. They were also used during this part of development to discuss ways that performance could be improved.

The most common topics at FORTRAN development meetings are COMMON blocks and interface development.

4.7.1.13. Library Structure

The library structure for Ada developments is much more complex than the library structure for FORTRAN developments[24]. There is one library for controlled source code (CMS library). The other libraries are maintained through the Ada compiler (ACS libraries). The ACS libraries contain object code, source code for automatic recompiles, files for module dependency tracking, and files for Ada's complicated library functions. These

libraries are hierarchical. When compilation is done in a sublibrary, the current library is searched for each required unit. If a required unit is not there, the parent library is searched.

The top level Ada Compilation System (ACS) library has global code for the system (e.g., utility package, global type package). The sublibraries on the next level are of three different types. One exists for each of the several subsystems of the dynamics satellite simulator, and some of these also had sublibraries. A library also exists for each developer, and then there are two more integration libraries. Coding and unit testing occur in the first two types of libraries. One integration library has code for the first two builds only (Build 0 + Build 1 = Release 1), and the other for all builds (Release 2). It was hoped that parallel development of each release would make development go faster, but it did not. Instead the problems of correctly maintaining two separate releases, which were each still changing, slowed things down, and increased difficulties. Since the library structure was much more complex with Ada than with FORTRAN, this caused library difficulties for the developers (e.g., compilation errors), as well as correctness difficulties (keeping two copies of Release 1 identical).

4.7.1.14. Call–Through Units

The Ada dynamics simulator contained many "call–through" units. A "call–through" unit is defined as one which contains only a call to another unit, and no procedural code. It exists so that a one–to–one mapping between objects in the design and code units is maintained. In other words, there is a one–to–one mapping between logical objects and physical objects. "Call–through" units can result when the design contains objects within other objects, perhaps several levels deep. This translates to packages inside packages. The other purpose for using "call–through" units is to maintain information hiding. It can be implemented with either library units or by nesting.

Using "call–throughs" on this project resulted in quite a bit of extra code. Estimates are that there are approximately 22K extra lines of code (carriage returns) from the extra specifications and bodies required. Thus there is that much more difficulty in code reading, testing, and other development phases due to the extra code. A simpler code structure would be more readable and probably more maintainable.

4.7.1.15. Use of Non–portable Features

Some non–portable features needed to be used for the sake of efficiency. For example, the hardware dependent floating point representation had to be used, rather than the software simulated one. In addition, the DEC screen management program was used to handle the displays. The team kept the non–portable features localized.

4.7.2. Code Reading

Code reading is done at the same time as unit testing. The developer who does the code reading for a particular unit is not the same one who developed the code.

The team found style errors more often than any other type. Other errors code reading helped to isolate were initialization errors, and incompatibilities between design and code. Logic errors were hard to discover in this application domain using either FORTRAN or Ada.

The types of errors found by code reading are different for FORTRAN and Ada. FORTRAN code reading finds wrong data types, calling sequence errors, and variable errors (declared but not used and used but not declared). For Ada, the compiler finds these. One developer felt code reading in Ada was not as interesting, because the compiler finds all the

interesting mistakes.

Some ways of using Ada made code reading more difficult. Heavy nesting or "call-through" units made the code harder to follow. Separate compilations tended to do this also. Each of these things can lead to having to look in multiple places to determine the correctness of functions. In addition, with nesting, it may not be clear exactly where to look. However, Ada's English-like style aided code reading.

Code reading was useful for learning to use Ada. The code from the GRO simulator was later used to help train another team in Ada developing dynamics simulators. In addition, when looking at each other's code, the developers saw new ways to handle problems and new algorithms for doing things. Code reading also helped to increase another developer's understanding of a different part of the system than he worked on.

4.7.3. Unit Testing

After developing his code, the same developer then tests it, while another does code reading. Once code passes both unit testing and code reading, the unit is put under configuration control (entered into the CMS library). Unit testing was more difficult than the team expected, and it was harder with Ada than with FORTRAN[18]. The methodology used was similar to the one used in FORTRAN developments.

4.7.3.1. Factors Complicating Unit Testing and Integration

Both unit testing and integration were complicated by several factors. One of these factors was the more complex library structure. Additionally, unit testing as well as integration was increasingly more difficult, the more levels of nesting there were present in the code. (See section 4.7.1.5, Library units vs. nesting [during coding] and section 4.7.4.4, Library units vs. nesting [during integration]). Though the definition of a unit was considered to be an object (package) for design purposes, for unit testing purposes a unit was viewed as a subprogram. This was probably because this was the way testing was done with FORTRAN. FORTRAN modules are isolated; the only major links between them are the global COMMONs. The Ada simulator had highly interconnected modules; that is, they depended on a lot of other code, and therefore are much more interdependent. Thus the team could not easily test most units in isolation, because each unit depended on too many others. (See section 4.6.2, Ada design documentation).

Usually with FORTRAN, little integration occurs at all until after unit testing. However, in this case, team members found it easiest to integrate up to the package level first, and then unit test. Integrated units were then tested choosing a subset of the possible paths at any one time, with the debugger.

A second difference was that no debug "write" statements were added to the code. This would have been too time consuming to recompile. For this purpose also, the debugger was used instead.

4.7.3.2. Debugger

The debugger was required for unit testing when nesting was used, because nested units are out of the scope of the test driver. Since it violates the usual visibility rules of Ada, the debugger is the only useful way to see nested units not named in the specifications. The debugger also had the advantage of allowing testing to continue without recompiling when problems were found (e.g., an uninitialized variable). Two other ways were tried to deal with the nesting during unit testing without much success. One way was to change the specifications of the outermost of the nested units so that the inner ones being tested could be seen from the outer level. The other way was to remove the inner piece of code to be unit

tested, and add the necessary context to it. Both ways were error prone and required a lot of time and recompilation.

4.7.3.3. Strong Typing

Strong typing was also somewhat of a problem with unit testing. It was more difficult to write test drivers, and the I/O was more complex. More operations and more code needed to be tested, especially due to the type proliferation. (See section 4.7.1.10, Strong typing). From the point of view of one FORTRAN programmer, different types that were still "just numbers" looked unnecessarily complicated.

4.7.3.4. Error Detection

After a clean Ada compile, the team felt more confident about correctness of the code than after clean FORT compile. The compiler finds many bugs like those usually found in code reading and unit testing with FORTRAN. Because of this, there is a tendency to be lazy with Ada. The tendency is to be overconfident in the fault detection abilities of the compiler and run-time system. Thus faults are overlooked. Some team members noted that the intuition for recognizing sources of failures with FORTRAN did not translate over to Ada.

Since several team members had not done a dynamics satellite simulator before, there was a problem determining if some of the mathematical calculations were correct. The mathematical specifications only give the algorithm, and not a range of reasonable inputs and outputs for the calculations. For those unfamiliar with the application, unit testing could not be completed until I/O values were provided.

4.7.4. Integration

Integration and integration testing were more difficult than the team expected, and more difficult than they were for the FORTRAN team. Each individual was responsible for integration and testing of his own subsystem. Then one individual was responsible for integrating the subsystems and integration testing for the whole release. Integration testing is functional. Typical integration problems during integration of a FORTRAN system are (1) performance (I/O, tasking, file allocation), (2) space, (3) interfaces, and (4) errors in flow of control. The User Interface, where almost all the tasks were, was the most difficult part of the Ada system to integrate.

4.7.4.1. Qualifications for Integration Tester

The team had problems which were the result of the integration tester being inexperienced in this application, and in development, generally. It was the integration tester who would identify the section of the system that was incorrect when tests were failed. The problem would be given to the developer whose code seemed to have the problem, and the problem might really be from somewhere else, in another developer's code. Exception handling (improperly done) is one thing that could do this.

4.7.4.2. Interfaces and Strong Typing

The team expected fewer interface problems during integration of the Ada code than they generally have during integration of FORTRAN. Early development of the interfaces led them to think this. But, the parameters changed a lot. In some cases, procedures were added. Some of these changes were related to the problem of developers not knowing whose procedure actually was supposed to perform certain functions (e.g., initialization, or conversions).

Strong typing was also a factor in the interface problems, and also made integration and integration testing more difficult. There was more code to test due to all the operations necessary to support all the types created.

4.7.4.3. Efficiency Issues

Late implementation changes are primarily done to improve efficiency. The design was intentionally created to simulate events as they occur in the actual hardware. The result however, was the recalculation of values many times which could have just been stored, since the values do not change often. Thus, the more realistic implementation was also less efficient, and this was discovered during integration.

In addition, direct access I/O was found to use a great deal of CPU time. Buffering was implemented to fix this problem.

Another inefficiency occurred due to the interaction of the DEC screen management system and task scheduling by the Ada run–time system. The run–time system enters the status of a task into a table when a task is elaborated (declared). Possible status entries are: (1) "ready", when the task has all resources it needs except the CPU, (2) "suspended" (waiting for rendezvous), (3) "waiting for I/O", (4) "terminated", and (5) "executing". These entries are used for scheduling; "ready" tasks are scheduled for CPU (by priority), and they execute until their time slice expires or status changes. When the DEC screen manager was being used, it would usually be waiting for input, and the associated task should have been marked as such; but instead, the run–time system marked it as "ready". The result was a grossly inefficient system, since the I/O task would take up a whole time slice waiting for I/O, every time it got the CPU. Tasks doing useful things got only a small percentage of the CPU. This was corrected by adding another task to change the entry in the table. The team had been advised to change the priorities of the tasks in order to fix the problem, but it did not work.

4.7.4.4. Library Units vs. Nesting

The team had intended to use nesting conservatively. During coding most team members thought they had done just that. But in retrospect, after integration and testing, many team members felt differently. Nesting had made their task more difficult. Nesting had been overdone, and library units not used enough.

For this project, library units went down about three or four levels, usually. Nesting went below that, sometimes as many as eight to ten levels[24]. While nesting had lots of problems associated with it, the only real disadvantage found with the use of many library units was the complicated library structure.

The team was surprised that nesting caused as much difficulty for them as it did. They had done a small project (5 – 6 K) to help them learn Ada when they were in training, where nesting had worked very well. But it did not scale up well. (Also see section 4.7.1.5, Library units vs. nesting [during coding]).

4.7.4.5. Exceptions

Errors and their sources can be obscured by using exceptions. This problem emerged particularly in later implementation, especially with integration. A scenario to demonstrate the problem follows. Suppose a particular procedure calls another unit, expecting some function to be performed, and certain kinds of data to be returned. If an exception is raised and handled in the called unit, and it is non–specific for the problem raising the exception (e.g., "when others"), the caller gets control back without the required function being performed. But the exception was handled and data was returned, so the call looks successful. Yet as

soon as the caller tried to use the data from the routine where the exception was raised and handled, it fails; yet another exception may be raised. Because of propagation, it can be very difficult to trace back the failure to the original source of the problem.

4.7.4.6. Tasking and Detecting Sources of Faults

Tasking was the most difficult feature to test, as one might expect. The type of testing done was functional. It was difficult with traditional testing methods to show that each task was actually invoked and worked right.

Much time during integration was spent in debugging tasks. While invaluable otherwise, the debugger was only of limited value with tasking. The worst example was with an array being passed by value instead of reference, causing a storage problem. Five tasks were deadlocked, and the task having the problem could not be localized. When stymied by this failure, the strategy finally used was to change tasks into sequential units, one by one, until the mistake was found. Though quite time consuming, this was required, due to the lack of diagnostics otherwise.

Isolating faults in tasks was complicated by exceptions, if they were present. They could terminate the task without indicating that they had done so. Since exceptions are not propagated except in the rendezvous, no tracebacks are obtained. Exceptions in tasks were commented out when problems arose in order to get a traceback. Exceptions raised at the rendezvous only provided tracebacks from the rendezvous on, and no previous information was given. If a task called a package not in its static scope, there was no exception propagation from the point of the failure in that package.

4.8. System Testing

The System Testing phase officially began July, 1987. One person was responsible for planning the system tests; this was finished during implementation. This same developer was the primary person putting time into doing system testing as well.

When implementation officially ended, the size of the project was 90K SLOC. Yet there were still parts of the system that had not been unit tested and put under configuration control. This was finished by December 1987. The new code from finishing unit testing, plus extra code to correct any failures found during system testing amounted to about 30K SLOC. The last 7–8K of code to total 128K was the Kalman filter, added April 1988.

Two reasons for the unusually large amount of implementation still being done during this phase are (1) other projects had higher priority for developers, and (2) a lack of analyst support (necessary to tell correctness of a number of units). System testing and the end of unit testing and implementation usually overlap in FORTRAN developments, but not to this large a degree. The usual reason for the coding and unit testing in this phase is a requirements change.

No acceptance testing was done for this project.

4.9. Phases – Overall

4.9.1. Size of Ada and FORTRAN Systems

The final size of the Ada system is 128K SLOC (carriage returns); the final size of the FORTRAN system is 44.6K SLOC. One third of the Ada system is specifications. More blank lines are also in the Ada system for readability, and more comments are also in the Ada code. One reason for more comments is that some of the constructs in Ada are more complex than any of the FORTRAN ones. Another reason the Ada system has more lines of

code is that many statements in Ada span multiple lines. In FORTRAN this is much less common. The style was regulated by the Ada Style Guide[40].

4.9.2. Reuse

The amount of reuse in each system is compared in Table 4.4. As with effort, estimates of reuse are made when the project begins; final results are also shown when known.

The Ada project reused a well-modularized section of FORTRAN code that was poorly documented (thus hard to build correctly from scratch), but known to work since it had been reused on other occasions.

The fact that the FORTRAN system actually reused less code than it planned on reusing is not unusual. During Requirements Analysis, code from old systems is reviewed once to find reusable pieces. This is not done again in later phases. Some of the code chosen as possibly reusable will turn out not to be suitable in the later phases of development.

4.9.3. Time Spent in Each Major Activity

Effort data is collected during a project in two ways. Total effort each week is recorded for each developer, manager and support person (e.g., clerical) working on the project. (See Resource Summary form in Appendix, Data Collection Forms). In addition, each individual records their activities, and the amount of time spent in each activity for each component worked on. (See Component Status Report form in Appendix, Data Collection Forms). The total effort each week gives the effort by phase, when broken down by phase dates (i.e., milestones). (See Tables 4.7 and 4.8). The hours in each activity (from Component Status Forms), regardless of when in the life cycle the activity was performed, gives the effort by activity. (See Table 4.5). Thus we have two different kinds of effort data tables (phase vs. activity) from two separate types of forms, though the names of the categories in each table are the same. The categories themselves are based on prior studies. The total hours of effort obtained from these two forms are close, but not the same. In the usual FORTRAN project, the primary activity carried on during a given phase gives the phase its name.

The Ada project took more effort than the FORTRAN project to complete, even when training time and acceptance testing time are not included. (Each of these activities was in only one of the projects). More time was also spent in each activity, except for requirements analysis. The time the Ada team spent rewriting the specifications (with CSM) was charged to training, though it might well be considered requirements analysis time. This extra time in every activity is contrary to what was expected, particularly for the later Ada phases.

The Ada project took more effort, partly because it was the first Ada project in this division. Learning, of course, takes extra time. It also took longer since there was essentially no reuse of code, and no reuse of design, while the FORTRAN project had a high level of reuse for both of these. Extra utilities had to be built that would be found in FORTRAN libraries as well. Finally, more functions were included in the Ada system than in the FORTRAN system; the Ada system had more functions beyond those required by the specifications.

4.9.4. Effort by Phase

The actual completion dates of each phase are given in Table 4.6. The usual phases are listed for the FORTRAN project. However, activities in the Ada development were distributed differently than activities in a FORTRAN development. As discussed earlier, one of the findings with the Ada project was much more overlap between the phases, because there was more overlap of the activities. Accordingly, two additional phases are included for Ada:

Design/ Code Overlap and Unit Test/ System Test Overlap.

In order to look at other data later (changes and failures) in terms of activity performed, as well as phases, we must approximate how much of each activity is in every phase. The phases – training, requirements analysis, design, code/unit test (implementation), and system test – are assumed to primarily consist of the activity they are named for, as they do with FORTRAN. The Design/ Code Overlap phase may well be considered primarily design, as far as activity is concerned. This is the period from CDR until completion of all parts of the design not finished before CDR, and completion of the Ada specifications and utilities (Build 0). Note that 6505 hours (from Table 4.5, the activity table) and 6870 hours (from Design + Design/Code Overlap, Table 4.7, the Ada phase table) are close.

Unit Test/ System Test Overlap is far more mixed, as far as activities during the phase. Table 4.5 (activity table) shows that the system test activity took a total of 3704 hours for the Ada project. Each overlap phase could be grouped with either the phase right before it, or the phase right after it. (See phase tables for Ada, Table 4.7, and for FORTRAN, Table 4.8). Since Design/ Code contained mostly design activity, it makes sense to combine this with Design rather than Code/ Unit Test (Implementation). This is not so simple with the Unit Test/ System Test Overlap phase. However, some approximations as to amount of each type of activity in the overlap phase can be made. The actual number of hours in this phase is 3319. (See Table 4.7). We can assume that the phase labeled as System Test is primarily system test activity. This is a reasonable enough assumption in this environment, according to past FORTRAN project data. Then if we combine hours from System Test with hours from Unit Test/ System Test Overlap, we get a total number of phase hours equal to 4816. It is possible to consider this as the System Test phase. 4816 – 3704 (number of system test activity hours) = 1112, or the number of hours of implementation activity (primarily unit test) actually done during the Unit Test/ System Test Overlap phase. Thus the rest of the hours in the Unit Test/ System Test Overlap phase are assumed to be system test activity. This amounts to 2207 hours of system test.

We conclude that approximately 1/3 of the Unit Test/ System Test Overlap phase is implementation activity, and 2/3 is system test activity. We note that the total number of hours that it took to complete the Ada project is not identical in Table 4.5 (activity table) and Table 4.7 (Ada phase table). However, the error is about 2/3 of one percent, and not significant for viewing general trends.

The FORTRAN project is fairly typical of FORTRAN projects from this division, except that the Acceptance Test phase is extra long, and more effort than usual was put into acceptance test activities. More revisions than usual were made in the FORTRAN system. (See Table 4.14, Reasons for changes: Acceptance Test phase).

4.9.5. Productivity

Some kind of "lines of code" measure is usually used to calculate productivity. But even though a comparison of sizes of the FORTRAN and Ada systems by lines of code may be of some interest, it is comparing apples and oranges for determining productivity. How much effort does a line of Ada code take compared to a line of FORTRAN code? What difference does kind of statement make? That is, how does effort for creating various types of declarations, or various kinds of executable statements vary?

The best way to compare productivity in our case seems to be to assume identical systems functionally (not entirely true, as explained elsewhere), and to compare the time it took to create the systems. The hours used will be from the Activity table in section 4.9.3 (Table 4.5). Since no acceptance test was done for the Ada project, and no training was done for the FORTRAN project, the time for these activities will be subtracted from each project.

Therefore, for FORTRAN we have: 15,164 - 2257 = 12,607 hours; for Ada we have: 22,966 - 2436 = 20,560 hours. But we also have to consider how much reused code was in each system. The FORTRAN system had 36% reuse, and the Ada project had 2%. It has been found in the SEL that reusing code costs about 20% of the cost of generating new code[28]. Thus we estimate that the FORTRAN system took 71% of the time it would take to create a new system like it, and the Ada system took 98% of the time it would take to create an identical system. Such a FORTRAN system would take 17,756 hours to complete (without acceptance test), and such an Ada system would take 20,980 hours (without acceptance test). The difference is 3224 hours.

4.9.6. Changes

Table 4.9 shows the reasons changes were made for the Ada and FORTRAN projects. This data is collected on Change Report Forms.[1] Whenever a change is made, the developer fills one out. Any records for changes are for units already under configuration control.

In either system, the primary reason for a change is to correct an error. The second most common reason in the FORTRAN system for a change is that a change had occurred in the requirements. Since development of the FORTRAN system was ahead of development of the Ada system, early requirements were more likely to have been implemented in FORTRAN, by the time a change in the requirements came.

The second most common reason for a change in the Ada system is a change to improve clarity, maintainability or documentation (e.g., readability of code, fixing comments, or changes to documents). The design methodology and its associated documentation was in a state of flux during this project. Since recording changes occurs after units have been under configuration control, this would not be the major reason for this. The team did put more effort into maintaining the documentation, and keeping it up to date than is usual for a FORTRAN project, since this was an experimental project. Improvement of user services is a close third when it comes to reasons for changes. The User Interface underwent many changes, and a large number of the extra, unrequired functions in the Ada system are in this subsystem.

Tables 4.10 through 4.14 show the reasons for changes that occurred in the different phases of each project. Just as errors and changes to the requirements were the top two reasons, respectively, for changes in the FORTRAN project overall, these were also the top two reasons for changes in every phase (after design). However, in System Test and Acceptance Test, the order for first and second position is reversed.

The most surprising thing about the changes for the Design/ Code Overlap phase was that there were so few changes, and none were error corrections (errors would have been discovered by failures). For the Ada system, from the Implementation phase on, error correction is the most common reason for changes. The period after the System Test phase is labeled Acceptance Test, even though there was no real Acceptance Test phase. During the Implementation phase, improving clarity, maintainability or documentation is the next most common reason for changes. In the Unit Test/ System Test Overlap phase, the second most common reason for changes shifts to requirements changes. By System Test, improving the User Interface, due to the tasks it contains, gave a lot of problems during integration and system testing.

We will now switch from considering change profiles by phase to considering change profiles according to activity. Changes were classified in Table 4.15 according to the activity

[1]See Appendix, Data Collection Forms.

being performed when the change was made. Several assumptions were made in order to derive activity data from the phase data. Assumption (1) is that, except for the Ada overlap phases, the major activity of each phase is the one the phase is named after. Former FORTRAN studies[28] have shown this is a reasonable assumption. Assumption (2) is related; the non–major activities of each phase are negligible. Assumption (3) is that the Design/ Code Overlap phase is design (discussed in section 4.9.4, Effort by phase). Assumption (4) is that the Unit Test/ System Test Overlap phase's changes (and likewise the error corrections in discussions to come) are randomly distributed throughout the phase.

Using these assumptions, the calculations for Table 4.15 were done as follows for the Ada project. The Design/ Code Overlap phase was combined with design. The Implementation phase (Code/ Unit Test phase) changes were added to 1/3 of the changes in the Unit Test/ System Test Overlap phase to get changes due to implementation activity. (See section 4.9.4, Effort by phase, where 1/3 of the Unit Test/ System Test phase is calculated to be implementation activity, and 2/3 is considered system test activity). The System Test phase changes were added to 2/3 of the changes in the Unit Test/ System Test Overlap phase to get changes due to system test activity. The post–system test phase ("Acceptance Test") was called acceptance test activity.

Unlike the Ada project, activity for the FORTRAN project was assumed to correlate with phase.

Table 4.15 shows that the percentage of changes during implementation (code/unit test) were similar for both projects, and higher for the Ada project during system testing. However, if acceptance test is ignored, which is not "normal" for either project, the FORTRAN project has an implementation to system test ratio of 76/24. The Ada project's ratio is 60/40. This is the opposite of what was expected, since the Ada development was expected to have more of its changes earlier in development.

Table 4.16 gives the number of changes in a particular phase divided by the number of hours in the phase. Even if Acceptance Test is excluded, the trend is for this normalized value to rise for Ada and decrease for FORTRAN as each project progresses. In addition, the FORTRAN changes/hours for any given phase is always lower than for Ada.

Another way to look at the changes is to consider how hard it was to isolate the change (See Table 4.17). For both FORTRAN and Ada, most of the changes were easy to isolate. On the average, an Ada change took slightly longer to isolate than a FORTRAN change. This is probably due to the newness of Ada here.

The averages were calculated by assuming a random distribution of the changes in each time category. Thus, for the "less than 1 hour" category, calculations were done assuming 1/2 hour for isolating every change. For "1 hour to 1 day", 5 hours were used for calculations. For "1 day to 3 days", 16 hours were used, and for "greater than 3 days", 32 hours were used. Unknown lengths of time for either isolating or completing changes were excluded from the calculations for average.

Table 4.18 shows the effort needed to make the changes required in each system. About 2/3 of the changes took less than one hour to complete for both systems. The Ada system had more changes (percentage) that took a day or less, while the FORTRAN system had a higher percentage of changes that took more than one day to complete. On average, a change in the FORTRAN system took much longer to finish than a change in the Ada system.

4.9.7. Failures

Intuitively, the Ada team thought the Ada project had about the same number of failures as the FORTRAN project had. But the Ada project actually had more failures. It

is impossible to say whether a higher percentage of the errors actually there was discovered by the Ada team, compared to the FORTRAN team.

Failures were classified in Table 4.19 according to the activity being performed at the time the failure was found and corrected. The same assumptions and methods for calculation are used here as were used in section 4.9.6 (Changes), for calculating changes by the activity being performed when the various changes were made. The totals on the failure tables are equivalent to those listed as "error correction" on the Change tables. Data for failures is likewise collected only after the units are under configuration control.

Table 4.19 shows that more of the FORTRAN failures (percentage) were found performing implementation (code/unit test) than was true for Ada. The Ada development found many more failures (percentage) during system test activities than the FORTRAN development did. This difference is even more exaggerated when acceptance test activities are excluded.

Table 4.20 shows the activity being performed when the error occurred, which led to the failure discovered later. This judgement is made by the developer correcting the problem. The majority of the time, this is the person who developed this section of code from design onward.

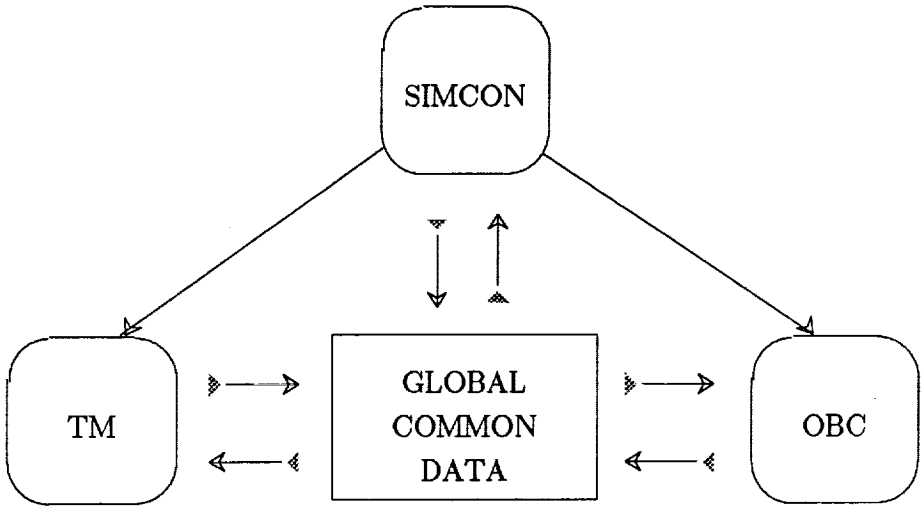
Coding errors are the largest source of failures in both systems. The percentage of coding failures is particularly high in the FORTRAN system. The Ada system had a significant number of failures due to design errors and to previous changes. This is not surprising since the Ada system had an entirely new design, whereas the FORTRAN system reused a significant percentage of past designs. Also Ada was new, thus developers were more likely to make errors when making changes, so failures from previous changes were higher.

Table 4.21 classifies failures according to type. The classifications mean what one might expect from the names, except for those that follow. "Computational" failures are due to problems with mathematical expressions. "Internal interface" refers to problems in module to module communications, and "external interface" refers to problems in communicating between modules in the system, and files or devices external to the system. For both the FORTRAN and Ada projects, data value or data structure problems and internal interface problems were the most frequent types of failures. The Ada team was surprised the rate of internal interface failures was as high as it was. However, part of the benefit of reused design for the FORTRAN team is that these are already partly worked out. Given this context, Ada came out quite well. The other major reason for failures in the Ada system was logic or control structure problems. The language was new, and the addition of concurrency would complicate this quite a bit.

Table 4.22 shows the effort needed to isolate failures in both systems. About 4/5 of the failures in the FORTRAN system took less than one hour to isolate. The Ada system had a little less than 3/5 in this category. This may be due in part to the compiler finding some of these before the code ever went under configuration control. (No data is recorded for a unit of code until the code is in the controlled library). The Ada system had a significant percentage of failures, about 1/3, that took between one hour and one day to isolate. On average, the Ada failures took much longer to isolate. The averages here were calculated the same way they were for the changes in the last section (section 4.9.6, Changes).

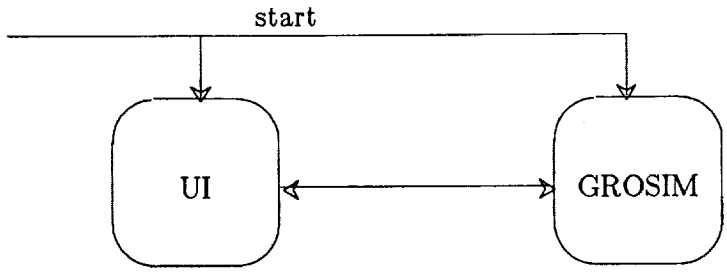
The effort required to correct the failures is shown in Table 4.23. The average length of time to correct these is similar for both projects. The FORTRAN project's average is slightly longer. The percentage of failures taking less than one hour to correct are similar for both FORTRAN and Ada. However, the Ada system had more failures that took up to a day to fix than the FORTRAN system did. The FORTRAN system had more failures that took over one day to correct.

Figures for Chapter 4

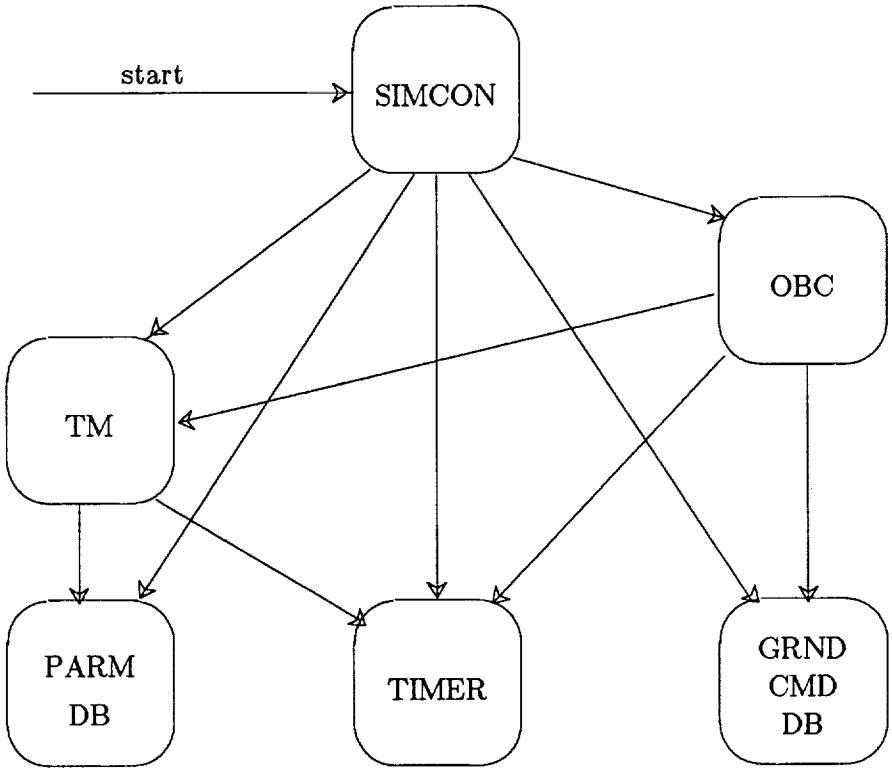


SIMCON	Simulation Control
TM	Truth Model
OBC	On–Board Computers

Figure 4.1. Top–level FORTRAN design.



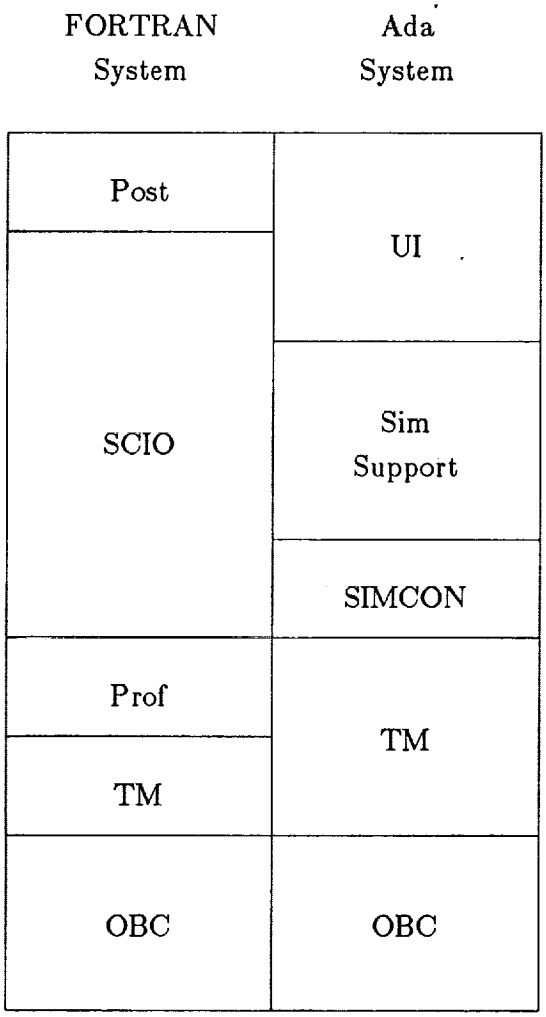
(a) Ada design — top level.



(b) GRO Simulator design

UI	User Interface
GROSIM	GRO Simulator
SIMCON	Simulation Control
TM	Truth Model
OBC	On–Board Computers
PARM DB	Parameter Database
GRND CMD DB	Ground Command Database

Figure 4.2. Top–level Ada design.



UI	User Interface
SIMCON	Simulation Control
TM	Truth Model
OBC	On-Board Computers
SCIO	Simulator Control and I/O
Post	Postprocessor Program
Prof	Profile Program
Sim Support	Timer, Parameter Database, Ground Command Database

Comparison of Subsystem Functions in Ada and FORTRAN Designs.
Figure 4.3.

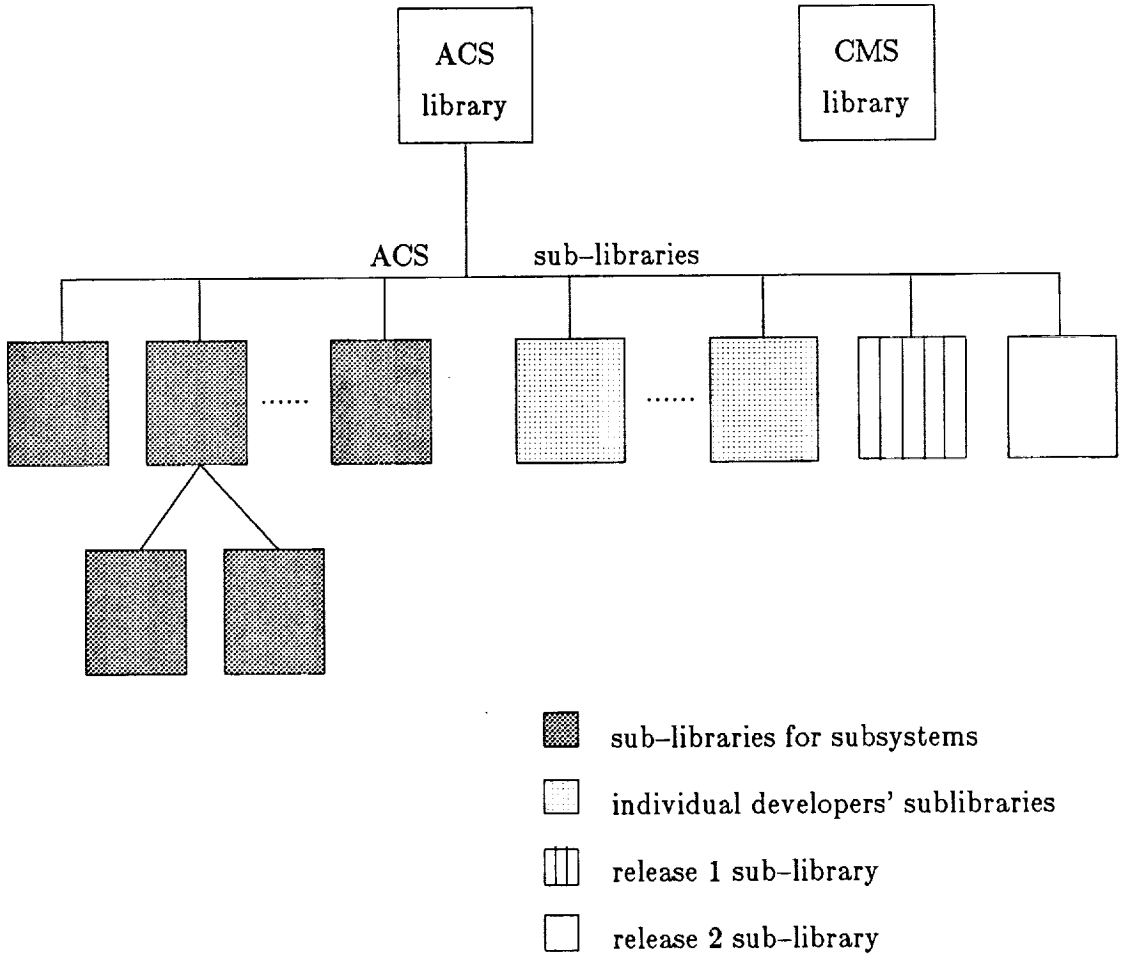


Figure 4.4. Ada Library Structure.

Tables for Chapter 4

	Estimate for FORTRAN*		Estimate for Ada*		Usual for SEL Across Projects
Requirements Analysis	1021	(10)	4620	(15)	(6)
Design	1531	(15)	12,320	(40)	(24)
Code/Unit Test	4083	(40)	7700	(25)	(45)
System Test	2552	(25)	4620	(15)	(20)
Acceptance Test	—		—		(5)
Other	510	(5)	1540	(5)	—
TOTAL	10,208**	(95)	30,800	(100)	(100)

Percentages are given in parentheses.

* Estimates were originally given in manmonths. A manmonth is assumed to be 22 days x 8 hours/day, or 176 hours/month.

** Estimates for each phase only account for 95% of hours in TOTAL.

Table 4.1. Estimated Effort by Phase for Each Project (In Hours).

	Ada		FORTRAN	
Requirements Analysis	6/29/85	(6)	3/15/85	(2.50)
End of Design	2/1/86	(7)	6/7/85	(2.75)
End of Code/ Unit Test	7/5/86	(5)	12/31/85	(6.75)
System Integration Test	10/4/86	(3)	4/31/86	(4)
Project Complete	12/6/86	(2)	9/30/86	(5)
TOTAL		(23)		(21)

Estimates were made when projects began.
Number of months is in parentheses.

Table 4.2. Estimated Project Completion Dates (by Phase). Calendar Time.

When Determined	Ada		FORTRAN	
	Estimate	Actual	Estimate	Actual
Before Project Starts	45K		41K	
Toward end of Design	90K			
Toward end of Code/ Unit Test		90K		
End of Project*	135K	128K	45K	44.6K

* Near end, for estimates.

Table 4.3. Size Characteristics (Total SLOC).

	Ada		FORTRAN	
	Estimate	Actual	Estimate	Actual
Reused Design	5	?	50	?
Reused Code	0	2	42	36

Table 4.4. Reuse Characteristics (Percentages).

	Ada		FORTRAN	
Training	2436	(10.6)	0	(0)
Requirements Analysis	680	(3.0)	1841	(12.1)
Design	6505	(28.3)	3361	(22.2)
Code	9671	(42.0)	5443	(35.9)
System Test	3704	(16.1)	1962	(12.9)
Acceptance Test	—		2557	(16.9)
TOTAL	22,996	(100.0)	15,164	(100.0)

Percentages are in parentheses.
Support staff hours (i.e., clerical) are not counted.

Table 4.5. Effort by Type of Activity (Staff Hours)

	Ada		FORTRAN	
Start Project	1/1/85		1/1/85	
Training	6/29/85	(6)	*	(0)
Requirements Analysis	9/7/85	(2.25)	2/10/85	(1.5)
Design	3/15/86	(6.25)	6/8/85	(4)
Design/ Code Overlap†	10/12/86	(7)		
Code/ Unit Test	6/27/87	(8.5)	12/28/85	(6.75)
Unit Test/ System Test Overlap	10/31/87	(4)		
System Test/ Integration	6/1/88	(7)	5/3/86	(4)
Acceptance Test	**	(0)	5/31/87	(13)
TOTAL		(41)		(29)

Number of months in each phase is in parentheses. Effort is not full time.

* No Training.

† Compilable specifications and utilities done during this phase.

** No Acceptance Testing done.

Table 4.6. Actual Project Phase Completion Dates.

	Estimate*		Actual		Usual for SEL Across Projects
Training	—		3346†	(14.6)	—
Requirements Analysis	4620	(15)	540	(2.4)	(6)
Design	12,320	(40)	2987	(13.1)	(24)
Design/ Code			3883	(17.0)	
Code/ Unit Test	7700	(25)	7291	(31.9)	(45)
Unit Test/ System Test			3319	(14.5)	
System Test	4620	(15)	1497	(6.5)	(20)
Acceptance Test	—		—		(5)
Other	1540	(5)	—		—
TOTAL	30,800		22,863	(100.0)	(100)

Percentages are given in parentheses.
Support staff hours (i.e., clerical) are not counted.

* Estimates were originally given in manmonths. A manmonth is assumed to be 22 days x 8 hours/day, or 176 hours/month.

† Project manager estimated 1000 of these hours are actually Requirements Analysis hours.

Table 4.7. Effort by Phase for the Ada Project (In Hours).

	Estimate*		Actual		Usual for SEL Across Projects
Requirements Analysis	1021	(10)	849	(5.6)	(6)
Design	1531	(15)	2830	(18.7)	(24)
Code/ Unit Test	4083	(40)	5397	(35.6)	(45)
System Test	2552	(25)	2315	(15.3)	(20)
Acceptance Test	—		3775	(24.9)	(5)
Other	510	(5)	—		—
TOTAL	10,208**		15,166	(100.1)	(100)

Percentages are given in parentheses.
Support staff hours (i.e., clerical) are not counted.

* Estimates were originally given in manmonths. A manmonth is assumed to be 22 days x 8 hours/day, or 176 hours/month.

** Estimates for each phase only account for 95% of hours in TOTAL.

Table 4.8. Effort by Phase for the FORTRAN Project (In Hours).

Type of Change	Ada		FORTRAN	
Error Correction	226	(42.2)	103	(39.6)
Planned Enhancemt	37	(6.9)	33	(12.7)
Implem. Requiremt Change	52	(9.7)	88	(33.8)
Improve Clarity or Documentation	88	(16.4)	15	(5.8)
Improvemt of User Services	67	(12.5)	8	(3.1)
Insert/Delete Debug Code	23	(4.3)	11	(4.2)
Optimz Space, Time or Accuracy	28	(5.2)	0	(0.0)
Adaptat to Environ	9	(1.7)	1	(0.4)
Other	5	(0.9)	1	(0.4)
TOTAL	535	(99.8)	260	(100.0)

Percentages are in parentheses.

Table 4.9. Reasons for Changes, All Phases.

Type of Change	N	DC	T
Planned Enhancemt	2	(50.0)	(0.4)
Optimz Space, Time or Accuracy	2	(50.0)	(0.4)
TOTAL	4	(100.0)	(0.8)+
TOTAL FOR ALL PHASES	535		
Percent Design/Code Phase to Total, All Phases			(0.7)+

N Number of changes.
DC Percentage of N using Design/Code overlap phase total.
T Percentage of N using Total for all phases.

Percentages are in parentheses.

* Phase dates are: March 15, 1986 to October 12, 1986 for the Ada project.
+ These do not agree due to round-off error.

Table 4.10. Reasons for Changes: Design/ Code Overlap Phase.

Type of Change	Ada*			FORTRAN†		
	N	I	T	N	I	T
Error Correction	99	(37.6)	(18.5)	65	(41.7)	(25.0)
Planned Enhancemt	28	(10.6)	(5.2)	30	(19.2)	(11.5)
Implem. Requiremt Change	17	(6.5)	(3.2)	39	(25.0)	(15.0)
Improve Clarity or Documentation	53	(20.2)	(9.9)	7	(4.5)	(2.7)
Improvemt of User Services	30	(11.4)	(5.6)	7	(4.5)	(2.7)
Insert/Delete Debug Code	14	(5.3)	(2.6)	8	(5.1)	(3.1)
Optimz Space, Time or Accuracy	15	(5.7)	(2.8)	0	(0.0)	(0.0)
Adaptat to Environ	5	(1.9)	(0.9)	0	(0.0)	(0.0)
Other	2	(0.8)	(0.4)	0	(0.0)	(0.0)
TOTAL	263	(100.0)	(49.1)+	156	(100.0)	(60.0)
TOTAL FOR ALL PHASES	535			260		
Percent Implem Phase to Total, All Phases			(49.2)+			(60.0)

N Number of changes.
I Percentage of N using Implementation phase total.
T Percentage of N using Total for all phases.

Percentages are in parentheses.

* Phase dates are: October 12, 1986 to June 27,1987 for the Ada project.

† Phase dates are: June 8, 1985 to December 28, 1985 for the FORTRAN project.

\+ These do not agree due to round-off error.

Table 4.11. Reasons for Changes: Implementation Phase.

Type of Change	N	UT/ST	T
Error Correction	66	(41.5)	(12.3)
Planned Enhancemt	3	(1.9)	(0.6)
Implem. Requiremt Change	28	(17.6)	(5.2)
Improve Clarity or Documentation	25	(15.7)	(4.7)
Improvemt of User Services	18	(11.3)	(3.4)
Insert/Delete Debug Code	2	(1.3)	(0.4)
Optimz Space, Time or Accuracy	10	(6.3)	(1.9)
Adaptat to Environ	4	(2.5)	(0.7)
Other	3	(1.9)	(0.6)
TOTAL	159	(100.0)	(29.8)+
TOTAL FOR ALL PHASES	535		
Percent Unit Test/System Test Phase to Total, All Phases			(29.7)+

N Number of changes.
UT/ST Percentage of N using Unit test/ System test overlap phase total.
T Percentage of N using Total for all phases.

Percentages are in parentheses.

* Phase dates are: June 27,1987 to October 31, 1987 for the Ada project.

+ These do not agree due to round-off error.

Table 4.12. Reasons for Changes: Unit Test/ System Test Overlap Phase.

Type of Change	Ada* N	Ada* S	Ada* T	FORTRAN† N	FORTRAN† S	FORTRAN† T
Error Correction	51	(53.1)	(9.5)	17	(35.4)	(6.5)
Planned Enhancemt	3	(3.1)	(0.6)	1	(2.1)	(0.4)
Implem. Requiremt Change	7	(7.3)	(1.3)	24	(50.0)	(9.2)
Improve Clarity or Documentation	9	(9.4)	(1.7)	2	(4.2)	(0.8)
Improvemt of User Services	19	(19.8)	(3.6)	0	(0.0)	(0.0)
Insert/Delete Debug Code	6	(6.3)	(1.1)	3	(6.3)	(1.2)
Optimz Space, Time or Accuracy	1	(1.0)	(0.2)	0	(0.0)	(0.0)
Adaptat to Environ	0	(0.0)	(0.0)	0	(0.0)	(0.0)
Other	0	(0.0)	(0.0)	1	(2.1)	(0.4)
TOTAL	96	(100.0)	(18.0)+	48	(100.1)	(18.5)
TOTAL FOR ALL PHASES	535			260		
Percent System Test Phase to Total, All Phases			(17.9)+			(18.5)

N Number of changes.
S Percentage of N using System Test phase total.
T Percentage of N using Total for all phases.

Percentages are in parentheses.

* Phase dates are: October 31, 1987 to June 1, 1988 for the Ada project.

† Phase dates are: December 28,1985 to May 3, 1986 for the FORTRAN project.

+ These do not agree due to round-off error.

Table 4.13. Reasons for Changes: System Testing Phase.

Type of Change	Ada*			FORTRAN†		
	N	A	T	N	A	T
Error Correction	10	(76.9)	(1.9)	21	(37.5)	(8.1)
Planned Enhancemt	1	(7.7)	(0.2)	2	(3.6)	(0.8)
Implem. Requiremt Change	0	(0.0)	(0.0)	25	(44.6)	(9.6)
Improve Clarity or Documentation	1	(7.7)	(0.2)	6	(10.7)	(2.3)
Improvemt of User Services	0	(0.0)	(0.0)	1	(1.8)	(0.4)
Insert/Delete Debug Code	1	(7.7)	(0.2)	0	(0.0)	(0.0)
Optimz Space, Time or Accuracy	0	(0.0)	(0.0)	0	(0.0)	(0.0)
Adaptat to Environ	0	(0.0)	(0.0)	1	(1.8)	(0.4)
Other	0	(0.0)	(0.0)	0	(0.0)	(0.0)
TOTAL	13	(100.0)	(2.5)+	56	(100.0)	(21.6)++
TOTAL FOR ALL PHASES	535			260		
Percent Acceptance Testing Phase to Total, All Phases			(2.4)+			(21.5)++

N Number of changes.
A Percentage of N using Acceptance Testing phase total.
T Percentage of N using Total for all phases.

Percentages are in parentheses.

* No true acceptance test phase for the Ada project. Data is for all changes after June 1, 1988.

† Phase dates are: May 3, 1986 to May 31, 1987 for the FORTRAN project.

+ These do not agree due to round–off error.

++ These do not agree due to round–off error.

Table 4.14. Reasons for Changes: Acceptance Testing Phase

Activity	Ada		FORTRAN	
Design	4	(0.7)		
Code/ Unit Test	316	(59.1)	156	(60.0)
System Test	202	(37.8)	48	(18.5)
Acceptance Test	13	(2.4)	56	(21.5)
TOTAL	535	(100.0)	260	(100.0)

Percentages are given in parentheses.

Table 4.15. Changes by Activity.

	Ada		FORTRAN	
Phase	Number	Norm.†	Number	Norm.†
Design/ Code	4	.001		
Code/ Unit Test	263	.036	156	.029
Unit Test/ System Test	159	.048		
System Test	96	.064	48	.021
Acceptance Test	13	.085	56	.015
TOTAL	535	.023	260	.017

† Normalization is based on number of hours in the relevant phase. Thus the units are: changes in the given phase per hour spent in the given phase. Total hours for whole project (including phases where no changes are possible) are used for TOTALs.

Table 4.16. Normalized Changes per Phase.

	Ada		FORTRAN	
< 1 hour	354	(66.2)	196	(75.4)
1 hour to 1 day	142	(26.5)	44	(16.9)
1 to 3 days	27	(5.0)	16	(6.2)
> 3 days	9	(1.7)	3	(1.2)
Not known	3	(0.6)	1	(0.4)
TOTAL	535	(100.0)	260	(100.1)
Ada	3.0 hours/ change (average)			
FORTRAN	2.6 hours/ change (average)			

Percentages are given in parentheses.

Table 4.17. Effort to Isolate Changes (All Phases).

	Ada		FORTRAN	
< 1 hour	348	(65.0)	167	(64.2)
1 hour to 1 day	149	(27.9)	51	(19.6)
1 to 3 days	29	(5.4)	28	(10.8)
> 3 days	6	(1.1)	12	(4.6)
Not known	3	(0.6)	2	(0.8)
TOTAL	535	(100.0)	260	(100.0)
Ada	3.0 hours/ change (average)			
FORTRAN	4.5 hours/ change (average)			

Percentages are given in parentheses.

Table 4.18. Effort to Complete Changes (All Phases).

Activity	Ada		FORTRAN	
Code/ Unit Test	121	(53.5)	65	(63.1)
System Test	95	(42.0)	17	(16.5)
Acceptance Test	10	(4.4)	21	(20.4)
TOTAL	226	(99.9)	103	(100.0)

Percentages are given in parentheses.

Table 4.19. Failures by Activity.

Origin of Failure	Ada		FORTRAN	
Code	131	(58.0)	92	(89.3)
Design	49	(21.7)	3	(2.9)
Functional Specifications	9	(4.0)	4	(3.9)
Previous Change	32	(14.2)	2	(1.9)
Requirements	5	(2.2)	2	(1.9)
TOTAL	226	(100.1)	103	(99.9)

Percentages are given in parentheses.

Sources of Failures (All Phases).
Table 4.20.

Error Class	Ada		FORTRAN	
Computational	28	(12.4)	12	(11.7)
Logic/ Control Structure	46	(20.4)	16	(15.5)
Data value or Structure	65	(28.8)	24	(23.3)
Initialization	29	(12.9)	15	(14.6)
External Interface	11	(4.9)	6	(5.8)
Internal Interface	47	(20.8)	30	(29.1)
TOTAL	226	(100.2)	103	(100.0)

Percentages are given in parentheses.

Table 4.21. Types of Failures.

	Ada		FORTRAN	
< 1 hour	130	(57.5)	84	(81.6)
1 hour to 1 day	81	(35.8)	15	(14.6)
1 to 3 days	11	(4.9)	3	(2.9)
> 3 days	3	(1.3)	1	(1.0)
Not known	1	(0.4)	0	(0.0)
TOTAL	226	(99.9)	103	(100.1)
Ada	3.3 hours/ failure (average)			
FORTRAN	1.9 hours/ failure (average)			

Percentages are given in parentheses.

Table 4.22. Effort to Isolate Failures (All Phases).

	Ada		FORTRAN	
< 1 hour	161	(71.2)	82	(79.6)
1 hour to 1 day	56	(24.8)	10	(9.7)
1 to 3 days	6	(2.7)	9	(8.7)
> 3 days	2	(0.9)	1	(1.0)
Not known	1	(0.4)	1	(1.0)
TOTAL	226	(100.0)	103	(100.0)
Ada	2.3 hours/ failure (average)			
FORTRAN	2.6 hours/ failure (average)			

Percentages are given in parentheses.

Table 4.23. Effort to Correct Failures (All Phases).

CHAPTER 5

Lessons Learned

5.1. Introduction

In this chapter we give a list of the lessons we learned during the course of the study. This chapter is organized into headings which match those in chapter 4, "Observations". Thus if the reader needs to refer back to the data for the context of a particular lesson, it will be easy to do so. This project helped us learn a great deal about the use of Ada with an application that is identical to the usual ones constructed in the Flight Dynamics Division at NASA/Goddard. It was not possible to fully answer all the questions originally posed with the data we were able to gather. Further case studies are needed to verify which effects are "first project" effects (especially due to a learning curve), and which are due to other things. In the final analysis, the question whether Ada or FORTRAN is better in this environment for these projects, must still wait for an answer.[1]

5.2. Effort and Size Estimates

(1) The Ada project actually took fewer manhours to complete than estimated. The project would surely still have been on schedule in this respect, even if acceptance testing had been done.

(2) The FORTRAN project took half again as many manhours as predicted. A lot of this is due to the unusually long time in acceptance testing with this project.

(3) Both the FORTRAN and Ada projects took more calendar time than estimated. The FORTRAN project took six months longer. The Ada project took eighteen months longer. The effort was not full time on either project.

(4) SLOC comparisons are not too useful when comparing the size or productivity of projects built in different languages. The effort to produce a line of code in one language is not comparable to the effort required in another. Moreover, within the same language, the level of effort may be very different for different types of statements (e.g., executable vs. non-executable statements, concurrent vs. sequential constructs, etc.).

(See also section 4.9.3, Time spent in each major activity, section 4.9.4, Effort by phase, and section 4.9.1, Size of Ada and FORTRAN systems).

5.3. Training

(1) Training in software engineering concepts, and not just language syntax was important to the subsequent success of the project. Success included the development of a truly new design, and an OOD methodology tailored to the Flight Dynamics environment. (See the sections on Preliminary design and Detailed design, 4.5, 4.6, 4.6.1, 5.5, 5.6 and 5.6.1).

---

[1]See Chapter 6, Answers to G/Q/M Questions.

(2) It is very useful to have access to an experienced Ada consultant.

(3) The training project (electronic mail system or EMS) was useful to help the team learn to work together with these new concepts, and get a little practical experience with Ada. However, the project also misled them as to the usefulness and applicability of certain features on the simulator. The EMS was still a small program, between 5 and 6K, and not related to the usual application domain. Certain features do not scale up well, such as nesting. Liberal use of nesting worked well on EMS, and not so well on the much larger simulator. Other features are much harder to control on large projects than on small ones, and the small training project gave no hint of a future problem. Strong typing was in this category.

Tasking was never used on the training project. Generics were more useful and less trouble in the simulator. The compiler needed to mature before it handled generics correctly.

(4) It was not until after working on a "real" project that the team really felt they had learned Ada sufficiently well to use it well.

(5) Ada training should include training in the Ada library structure, due to its complexity. (See section 4.7.1.13, Library structure).

(6) Training for managers, and not just developers is important. (See also section 4.6.2, Ada design documentation).

5.4. Requirements Analysis

(1) Ada design issues began to surface during the Requirements Analysis phase.

(2) The Composite Specification Model (CSM) succeeded in removing the preliminary FORTRAN design embedded in the original specifications document (called "Specifications and Requirements Document").

(3) Rewriting the specifications using the CSM helped the development team have a much better understanding of the requirements. This was a useful way to develop greater familiarity with the application.

5.5. Preliminary Design

(1) The Ada team found that tailoring an OOD methodology to their particular corporate environment's needs was essential. This led the team to develop their own methodology (GOOD).

(2) Some of the issues to consider when choosing and tailoring a design methodology include: type of application (sequential vs. a high degree of concurrency), real-time or not, field of application (scientific, business, others), and research vs. production environment.

GOOD worked well for a scientific, mostly sequential, non-real time production-type project, such as this simulator.

(3) Both graphical (object diagrams) and textual (object descriptions) ways to represent the design fully were required. This was much clearer than using one type of design representation only.

5.6. Detailed Design

(1) The tailored design methodology, GOOD, was chosen to continue with into detailed design. It was the best suited to the application and needs of the environment.

5.6.1. Comparison of the Ada and FORTRAN Designs

(1) The Ada team actually did produce a different design than the FORTRAN team had for the same application. Partially, this was due to not having a real–time requirement on the Truth Model (TM) subsystem, as the FORTRAN team did. It was also due to understanding software engineering principles, and principles behind OOD. For example, the more realistic model of the satellite which was reflected in the Ada design stems from this.

(2) The degree of coupling that resulted between modules was surprising. The units are tied to each other in complex ways, rather than only through global COMMONs. (See also section 5.7.3.1, Factors complicating unit testing and integration).

5.6.2. Ada Design Documentation

(1) The design methodology and its representations should be set before the start of a production project. Otherwise, miscommunication will result between developers and managers. Extra time will also be required to update the design, when representations change. But this is also part of the cost of tailoring a design methodology to a new environment with an initial project. The benefits from this are expected on future projects.

(2) Automated tools to aid in maintaining the design documentation would help a great deal. A lot of work is required to develop and maintain the object diagrams and object descriptions.

(3) The design notation should be amended to include the control interactions required by tasks, if tasks are to continue to appear in future projects. (See section 4.7.1.6, Concurrency and tasking).

(4) Managers need to understand the design methodology and its notation, not just developers. This understanding is crucial for communication at reviews such as PDR and CDR, and to allow evaluation of the progress to that point. Training managers in OOD and software engineering principles, from a management perspective, will accomplish this.

(5) The design notation is not so easy to understand, if the design methodology is not understood, and the philosophy behind it.

5.6.3. Timing of Reviews and Phase Boundaries

(1) Phases did not abruptly start and end, but rather gradually moved from one to the next. This is true with FORTRAN phases. While each is named for their primary activity, all activities go on to some degree in every phase. It *appeared* however, to the developers to not be so, for FORTRAN, and yet to very much be true with Ada.

The reasons FORTRAN developments usually appear to have distinct phases is due to well defined milestones appearing at the end of most phases. Where a milestone does not exist (e.g., between Implementation and System Test), there still appears to be a distinct dividing line. The line between Implementation and System Test comes when all unit testing is finished (except for units generated be requirement changes), and subsystem integration is done.

However with the Ada project, two phenomena occurred. The first is that the milestones for the phases did not appear where the team subjectively felt they should. The second is that activities actually had different definitions, and this made things seem far less clear. For instance, the use of Ada specifications has aspects which traditionally (to a FORTRAN mindset) are attributed to both design and code. Thus

determining which phase the development of the Ada specifications belongs to, meant becoming conscious of assumptions from the FORTRAN development legacy, and changing them appropriately.

(2) The CDR should be held later in development, after compilable PDL can be included, and the types are developed. This was felt by the team not only to be a design activity, but also it would increase confidence in the correctness of the design. It requires early development of types and interfaces. (See also section 4.7.1.8, Interface development, and 4.7.1.10, Strong typing).

5.7. Implementation

(1) The possibility of doing bottom-up implementation should be considered. (Design would still be top-down). Given the way Ada dependencies work, this might prove an easier way to unit test and integrate.

5.7.1. Coding

5.7.1.1. Builds

(1) The more general utilities and the application specific utilities should not be in the same package. This would separate out the reusable pieces from the non-reusable ones.

(2) Compilable PDL during design would have been a desirable feature. It has several benefits. Type checking and interface checking done at an early stage can increase confidence in the design. This requires more detailed planning at an earlier stage than is usual in a FORTRAN project. The benefit is tempered somewhat when interfaces are not localized to minimize the effect of changes.

5.7.1.2. Coding Issues and Standards

(1) Developers discovered from experience that a unit which calculates a value once, and which acts as a constant ever after, should be coded as a function in the declaration section of the code.

5.7.1.3. Effect of Design on Implementation

(1) All features except tasking were easily coded from the design documents, for those familiar with OOD. The team considered the transition to Ada code from the OOD design easier than the transition to FORTRAN code from the FORTRAN design.

5.7.1.4. Design Additions and Changes

(1) The Ada and FORTRAN projects both had design additions. More changes to the existing design were done in the Ada version, because of inexperience and the learning curve accompanying a first project, and because the Ada project was experimental and thus did not have the time crunch.

(2) Design "additions" involving constructs such as tasking, which are more powerful than any in FORTRAN, may be viewed as changes, and must have the appropriate level of consideration given to them.

5.7.1.5. Library Units vs. Nesting

(1) Nesting had the following disadvantages: (1) it increased recompilation costs, (2) reading the code and tracing problems was more difficult than it was with library units,

and (3) it made reuse harder. It was harder to uncouple unnecessary code when it was nested than when library units were used. (4) Nesting also made unit testing more difficult. Library units did not have any of these disadvantages. (See section 4.7.3, Unit testing).

(2) Two or three developers were brought onto the Ada project only for implementation. The high degree of nesting made coming onto the project more difficult, because it was harder to locate particular procedures in the code, than it would have been with library units. Despite this, it was felt that it took less time to bring on new staff on the Ada project, than it does on FORTRAN projects.

(3) Library units had the one disadvantage of making the library structure more complex.

(4) Library units had many advantages while nesting had few advantages, and many disadvantages on a project of this size. For these reasons, using library units often and nesting sparingly is recommended. (See also section 5.7.4.4, Library units vs. nesting [during integration]).

5.7.1.6. Concurrency and Tasking

(1) Problems in this area were due to the inherent difficulties with concurrency, and not with Ada tasking itself. In fact, the tasking construct makes concurrency so easily available, that it is easy to overuse this feature. More care and restraint is required on the part of the developers therefore, to make sure they plan to use it in a manner producing correct programs.

(2) All tasks in the system and their interactions should be planned during design. A global analysis and overview of the system's tasks should be prepared as part of the design documentation. This should prevent task proliferation. In this project, the two major reasons the number of tasks grew were correction of failures (e.g., deadlock), and incomplete consideration of the place a given task should have in the system.

(3) Functions that are being considered for tasking need to be carefully considered, to determine whether concurrency will really provide results superior to sequential processing.

(4) The minimum number of tasks required, and the simplest possible design for these should be used, due to correctness difficulties with concurrency. The team felt in retrospect that sparing use of tasks is very important. This type of application has little need for concurrency anyway.

(5) Another advantage from developing tasks fully during design is that the more experienced personnel will be designing them. This is appropriate for more difficult and critical parts of a system. Implementation is more likely to have junior developers.

5.7.1.7. Generics/ Separate Compilations

(1) These were easy to use and helped make code in the system more manageable.

(See also section 5.7.1.13, Library structure).

5.7.1.8. Interface Development

(1) Effects on interfaces were not localized in one respect. Changes to global types, which occurred a lot during the project, would affect many interfaces and require interface changes.

(2) Type changes were one of the most common reasons for interface changes. Another common reason was parameter changes. This is related to some of the problems the

team had in knowing whose components performed various functions (e.g., initialization, and explicit conversions).

5.7.1.9. Global Types

(1) Recompilation is more of a problem when there are a lot of global types, because any change to one of the types requires recompilation of a large part of the system.

(2) More interfaces are affected by a given change when many global types are used.

(3) Types should be placed at as low a level in the design of the system as possible, to reduce the number of units dependent on them. The global types package should be as small as possible.

(4) A global types package makes reuse more difficult, since it adds more context.

5.7.1.10. Strong Typing

(1) This feature caught some types of faults much earlier than they are necessarily caught with weakly typed languages such as FORTRAN.

(2) Type proliferation became a serious problem during coding. During late design, the team realized to some extent that a problem existed with the types, but no one realized then that the problem would yet become a lot worse. To overcome this, a data type analysis needs to be added to design. This would limit complexity stemming from a large number of base types. Subtypes of these types can be used in various parts of the application.

5.7.1.11. PDL and Prologs

(1) Algorithms are more helpful than just descriptions in the prologs. They are more exact.

5.7.1.12. Meetings

(1) More meetings were required initially with the new technology to help team members educate each other about things they learned during the project.

(2) More meetings were required to deal with things incompletely specified in the design such as which units initialized variables, or performed conversions. Less confusion would result if the design specified functions down to the procedure level.

(3) Since recompilations of the system were very slow, meetings were important to warn developers to plan for an upcoming recompilation.

(4) Recompilations were generally done overnight. Because of this, and since meetings did not always succeed in warning the developers, strategies were invented to avoid being surprised by the need to recompile the code a developer planned to work with. Routines the developer knew he would need were saved from the controlled library into his own library before changes to the controlled library were made. Then he would have the old versions of code, and he could temporarily avoid having to recompile his code, which was dependent on the old code he copied from the controlled library. This short term solution to avoid recompilations and save time worked fairly well.

5.7.1.13. Library Structure

(1) It took the team a little while to get accustomed to the more complex structure of an Ada library.

(2) Parallel testing of Release 1 and Release 2 required maintaining two copies of Release 1 in two separate libraries. This was not worth the overhead required, and slowed things down rather than speeding them up.

(3) Testing required many stubs, due to the top–down nature of implementation, and the high degree of coupling between modules. Having a library of specifications ready when implementation starts would make development easier. These specifications can stand in (as stubs) for the units they specify during compilations, when other pieces of code refer to them. This is also one of the benefits of being able to separate specifications from bodies. The system structure can be set up early.

5.7.1.14. Call–Through Units

(1) These should be used sparingly, since all the specifications required to implement them increase the code size, and thus code reading and testing are harder. Logical and physical objects should be treated differently. Logical objects exist in the design; physical objects exist in the code.

5.7.1.15. Use of Non–portable Features

(1) For the sake of efficiency, non–portable features were used, however they were kept localized.

5.7.2. Code Reading

(1) The emphasis should be different when code reading Ada than when code reading FORTRAN, because different faults are found with each. The types of faults found in code reading with FORTRAN are often the same ones as the compiler finds with Ada.

(2) The team trusted the correctness of the Ada code syntactically and semantically more than the FORTRAN code, since the Ada compiler catches so many more faults than the FORTRAN compiler does. This had the psychological effect of making code reading seem less important with Ada. This extended somewhat to trusting the correctness in areas the complier cannot check, such as flow of control and logic.

(3) The most common problems found by Ada code reading were style problems.

(4) Ada is not automatically more readable. This depends on several style elements. Notably, nesting and "call–through" units can decrease readability.

(5) Code reading is a useful tool for teaching Ada. A second Ada team started another simulator project sometime after this one was well underway. That team found that reading the code from this project was very instructive.

5.7.3. Unit Testing

(1) Unit testing was harder than expected, and harder with Ada than with FORTRAN.

5.7.3.1. Factors Complicating Unit Testing and Integration

(1) Unit testing and integration were made more difficult by the following factors. (1) As the degree of nesting increased, the difficulty of testing increased. (2) The module–to–module coupling was higher with the Ada system, and made testing harder. (3) Ada's more complex library structure made testing harder also.

(2) Tasking, strong typing, exception handling and nesting are the Ada features which caused the most difficulty during unit testing and integration on this project. The interaction of Ada features such as exceptions and tasking, which were new and complex in their own right, caused many more difficulties for the team than they would

have had otherwise. Part of the problem is the fact that exceptions behave differently in tasks than in other kinds of units.

(3) Unit testing with Ada should be done differently than with FORTRAN. Alternating integration and unit testing worked best. A whole package or small subsystem should be considered a unit in an Ada system, rather than a subprogram being considered a unit, as it is in a FORTRAN system.

(4) Unit testing is best done without making any changes to the code, thus avoiding any recompilations. For example, adding "write" statements is not a good idea. (See section 5.7.3.2 also, Debugger).

5.7.3.2. Debugger

(1) A debugger is required for doing unit testing and integration testing without recompilations.

(2) The necessity for a debugger increases as levels of nesting increase.

5.7.3.3. Strong Typing

(1) More code had to be tested, since there were more operations in order to deal with the increased number of types, and the I/O for each of these. Test drivers needed I/O routines for each type. Controlling type proliferation through abstract data type analysis should have a positive effect here, as well as in coding.

5.7.3.4. Error Detection

(1) The intuitions for finding errors did not translate over from FORTRAN to Ada. New intuitions had to be developed.

(2) Since the compiler and Ada run-time system catch so many faults, there was a tendency to over-rely on these, and not to regard code reading as being as important for Ada as it is for FORTRAN. (See section 5.7.2, Code reading).

(3) For individuals unfamiliar with the application, the algorithms from the mathematical specifications were not enough to determine correctness of some of the mathematical units. If reasonable I/O values were also provided, the developers could determine correctness of these units independently of the analysts.

5.7.4. Integration

(1) Integration and integration testing were more difficult than the team expected, and more difficult than with FORTRAN.

5.7.4.1. Qualifications for Integration Tester

(1) In order to pinpoint the section of code giving problems correctly, the integration tester needs to be a person with both development and application experience. This experience will help the individual determine the source of problems arising during the tests.

5.7.4.2. Interfaces and Strong Typing

(1) The team had more interface problems than they expected. Partly, this is due to having a new design. Strong typing and parameter changes also contributed to the problem.

5.7.4.3. Efficiency Issues

(1) Some important inefficiencies in the Ada system were due to modeling reality too closely. This is particularly true in the simulation cycle, where calculations are done over and over even though the values have not changed, or have changed only an insignificant amount.

(2) The DEC screen management package and task scheduling in the run–time system interacted inefficiently with each other. The CPU was left idle for large amounts of time.

5.7.4.4. Library Units vs. Nesting

(1) Although the team thought they were using nesting conservatively, after unit testing and integration, they decided they had not.

(2) It was a surprise to find out that nesting works well on small projects, such as the training project, but not on larger ones.

5.7.4.5. Exceptions

(1) Exceptions should be developed as an integral part of the abstractions created in design, and not an "add on" during implementation.

(2) Well–coded exception handlers helped a lot in locating faults, while badly coded exception handlers hindered finding faults.

(3) For every exception, the design should show (1) what exception would be raised, (2) where it will be handled, and (3) what should happen.

5.7.4.6. Tasking and Detecting Sources of Faults

(1) Errors involving tasks were the most difficult to find and correct. Tools (e.g., debugger) and methods used for finding faults in sequential code were of little use.

(2) One of the major hindrances to integration was getting tasks to interact properly.

(3) Exceptions and tasks interact in some ways a novice Ada user would not expect.

5.8. System Testing

No lessons.

5.9. Phases – Overall

5.9.1. Size of Ada and FORTRAN Systems

(1) The Ada system was larger due to (1) more lines of code per construct (due partly to the nature of Ada, and partly to the style adopted), (2) specifications (1/3 of the system), (3) use of "call–through" units, (4) more blank lines, and (5) more comments. Some of the additional comments were extra explanation accompanying complex constructs.

(2) The final size of the FORTRAN system was approximately the same as the predicted size. The final size of the Ada system was almost three times the predicted size. How much larger the Ada system is than the FORTRAN system depends on how you measure size (e.g., SLOC or statements).

(3) The number of executable statements in each system is approximately the same.

(See also section 5.2, Effort and Size Estimates).

5.9.2. Reuse

(1) The Ada team found that integrating FORTRAN code into the Ada system was easy to do.

(See also section 5.7.1.9, Global types, and section 5.7.1.5, Library units vs. nesting).

5.9.3. Time Spent in Each Major Activity

(1) The Ada project took longer than the FORTRAN project overall, and also took longer in every activity except requirements analysis activities. Factors affecting this are (1) no design reuse, (2) little code reuse, and (3) the time it takes for learning on a first time project. These would add time to every activity except the requirements analysis ones.

(2) In relative percentage of time spent in each activity, the Ada and FORTRAN projects were similar. It was expected for Ada to take longer in design, and less time in implementation and testing. Instead, Ada took most time in the implementation activities. Secondly was design, and thirdly system test.

5.9.4. Effort by Phase

(See also section 5.2, Effort and size estimates, which has some lessons related to actual, overall effort vs. the estimates made).

(1) The Design/ Code Overlap phase is primarily design activity. The design was finished, the Ada specifications entered into the system, and the system utilities completed during this phase.

(2) The Unit Test/ System Test Overlap phase is estimated to be 1/3 implementation activity and 2/3 system test activity. There is much more overlap of these activities in the Ada project than is usual for FORTRAN projects.

The actual phase divisions for each project is shown in Table 5.1. The overlap phases have been adjusted. Design/ Code is combined with Design, and Unit Test/ System Test is split between Implementation and System Test. For comparison, the usual amounts of time FORTRAN projects spend in each phase is included.

	Ada	FORTRAN	Usual in SEL
Pre–design†	17	6	6
Design	30	19	24
Implementation	36	35	45
System Test	17	15	20
Acceptance Test	*	25	5

† Includes training for Ada personnel.

* No Acceptance Test was done.

Table 5.1. Time per Phase (Percentage).

5.9.5. Productivity

(1) Productivity figures based on SLOC are not meaningful, since "SLOC" has different meanings for different languages.

(2) The FORTRAN system took about 85% of the effort that the Ada system took. This is based on effort data only, and has nothing to do with system size. It assumes the same functionality for both systems (the Ada system actually had a bit more), and the effort data is adjusted to reflect what effort would have been, if there were no reuse.

5.9.6. Changes

(1) Error correction is the primary reason for changes in either the FORTRAN or the Ada systems, at any time in the life cycle (after code goes under configuration control).

(2) Twice as many changes were made in the Ada system as in the FORTRAN system.

(3) There are several types of changes that appear at different frequencies for one system, compared to the other. The FORTRAN system had a much higher percentage of requirements changes to implement; the Ada system had a significantly higher percentage of changes involving documentation and also a significantly higher percentage of changes involving user services (this involved the User Interface).

(4) If we look at the percentage of each type of change within each phase, we see the following. For Ada, the percentage of changes due to error correction rose as we progress through the phases. For FORTRAN, the percentage of changes due to error correction decreased in each phase after the implementation phase.

(5) For both systems, the percentage of changes in the "error correction" category, compared to the total number of changes made throughout the project, decreases significantly after implementation. Most changes of any sort are made during implementation. This is true for both projects. In fact, if 1/3 of the changes from the Unit Test/ System Test Overlap phase are included with Implementation for the Ada project, 60% of the changes made in both systems is done during Implementation.

(6) The FORTRAN project had many requirements changes occur even through Acceptance Testing. The Ada project had few changes for this reason after System Test began. The requirements were well determined by the time the Ada team reached this point in the project.

(7) 60% of Ada's documentation changes were made during implementation; just under 50% of FORTRAN's documentation changes were made then, and nearly all the rest were made during Acceptance Testing. Six times as many changes of this type were made overall in the Ada system as compared to the FORTRAN system.

(8) FORTRAN's changes to improve user services were nearly all done during implementation; Ada's changes to improve user services were well distributed throughout the life cycle. Eight times as many changes of this type were made overall in the Ada system as compared to the FORTRAN system.

(9) If Acceptance Test is ignored (it is not "normal" for either system), the FORTRAN project had about 76% of its changes done during implementation activities, and about 24% done during system testing activities. The Ada project had about 60% of its changes done during implementation activities, and about 40% done during system testing activities. Thus, the FORTRAN project had more changes made earlier in development than the Ada system did. This is contrary to expectations.

(10) Changes were easy to isolate in either system.

(11) Less time was spent making changes in the FORTRAN project, than was spent making changes in the Ada project. (See Table 4.16, Normalized changes per phase).

(12) An average change in the FORTRAN system took much longer to finish than an average change in the Ada system. This does not contradict the point noted just above, since there were many more changes made in the Ada system.

5.9.7. Failures

(1) If Acceptance Testing is ignored, the FORTRAN project had about 79% of its failures found during implementation activities, and about 21% found during system testing activities. The Ada project had about 56% of its failures found during implementation activities, and about 44% found during system testing activities. This appears, then, to say that the FORTRAN project corrected more errors sooner. However, we must remember that we only have data for faults found after code went under configuration control. It may be that the Ada compiler found faults which were then removed, and therefore are not in the data. The FORTRAN team may have found these same kinds of problems after the code was under configuration control, and therefore they are counted in the FORTRAN project. Given these assumptions, if the data for the Ada and FORTRAN projects included everything and not just data after configuration control, the Ada percentages might be more like the FORTRAN percentages. These assumptions also imply, however, that the ratio of Ada failures to FORTRAN failures would be even higher than in the current data. But it is also true that these types of faults are extremely easy to correct for Ada, so this does not seem important.

(2) Coding errors is the major reason given (by far) for sources of failures in both systems.

(3) In the Ada system, many more failures were due to design errors and to prior changes to the system, than was the case in the FORTRAN system. This can be expected, due to a brand new design, and the newness of Ada, respectively.

(4) There were a bit more than twice as many failures corrected in the Ada system, compared to the FORTRAN system.

(5) The distribution of types of failures is similar for both systems.

(6) Isolating the source of failures took significantly longer for the Ada project than for the FORTRAN project. Since the Ada compiler finds mistakes not found in FORTRAN, some of the easiest faults may not be left to find in the Ada system.

(7) Failures took about the same length of time to correct in both systems.

CHAPTER 6

Answers to G/Q/M Questions

This chapter will indicate what the answers are to the questions posed in section 1.3, and give cross–references in the text for the answers.

I. Process and Product Conformance (Characterize the development methodologies, and resulting product)

(1) What was the overall process model applied during the Ada development, including the processes applied within each phase of development?

The prescriptive Ada development model was a modified version of the standard FORTRAN development model. The modifications were things such as longer design and shorter test phases, which were in accord with the usual expectations for an Ada development. In addition, modifications allowed for various experiments with design methodologies. (See 3.2).

(2) What was the process applied during the standard FORTRAN development, including the processes applied within each phase of development?

The standard FORTRAN development used a form of the waterfall development methodology (See 3.1).

(3) How well did the Ada developers understand object–oriented design, and the principles behind it?

The Ada team applied OOD in such a way as to create a truly new design for their dynamics satellite simulator (See 4.6.1 and 5.6.1). They were also able to develop their own development methodology, tailored to the environment of the Flight Dynamics Division at Goddard (See 4.5 and 5.5). This methodology is now used with other Ada projects.

(4) How well did the Ada developers know Ada?

This was the first Ada project for everyone originally on the team (See 3.3). For implementation, some individuals were added to the team who had worked on one prior Ada project, but were not familiar with the design methodology used here. That application was also very different from this one. The team felt that it took working on a production–type project to learn Ada well enough to use it well on future projects (See 5.3).

(5) How well were the processes applied, which were used during the Ada and FORTRAN developments?

These were the same methods used many times before for the FORTRAN development, however it was new to apply these to an Ada development (See 3.3). Because of new issues with Ada, there were problems applying all the FORTRAN processes without modification. These processes are: analyzing requirements (4.4, 5.4), design (4.5, 4.6, 5.5, 5.6), coding (4.7.1, 5.7.1, and all subsections), code reading (4.7.2, 5.7.2), unit testing (4.7.3, 5.7.3, and subsections, particularly 4.7.3.4), and integration and integration testing (4.7.4, 5.7.4, and subsections). Defining the processes themselves were some of

the problems which arose with Ada (See 5.6.3).

(6) How was the training done for Ada?

It was important to address software engineering principles, as well as language issues, and OOD methodologies. Training included Booch's OOD methodology, Cherry's Process Abstraction Method (PAMELA), Alsys videotapes, and an electronic mail system as a training exercise (See 3.2 and 4.3). Some Ada features work well on small projects, but do not scale up. In retrospect, the simulator project itself actually acted as a training project, which was in the usual application domain, because it was a first project of its type. In addition, training for managers, and not just developers, is important.

(7) How were specifications represented for Ada and FORTRAN?

Specifications are functional and contain high level FORTRAN design (See 3.2). The Composite Specification Model (CSM) was used to rewrite the requirements and eliminate the FORTRAN biases for Ada development (See 4.4 and 5.4).

(8) How well do certain design methodologies work with Ada?

Preliminary design was done with three OOD methodologies to discover which one worked best in this environment (Booch's methodology, PAMELA, or GOOD). (See 4.5 and 5.5). The team's own methodology, General Object Oriented Design (GOOD), was chosen to use for detailed design (See 4.6 and 5.6).

(9) How was the product documented for both Ada and FORTRAN?

Object diagrams and object descriptions were used to represent the Ada design (See 4.5 and 5.5). Issues related to these representations are discussed in 4.6.2. (See also 5.6.2). Program design language (PDL) and prologs were used for both FORTRAN and Ada. FORTRAN designs are done with structure charts as part of the structural decomposition development methodology (See 3.1 and 3.3).

(10) How were implementation and testing done in FORTRAN and Ada?

For both projects implementation was done top–down, and the implementation plan was based on builds (See 4.7 and 4.7.1.1). Code reading had a different emphasis with Ada (See 4.7.2 and 5.7.2). With FORTRAN, within each build, unit testing is done first, and then integration. But this approach to unit testing did not work for Ada (4.7.3, 5.7.3, and subsections, especially 4.7.3.1), partly due to a higher degree of coupling in the Ada system (5.6.1), and the high degree of nesting (4.7.1.5). The Ada team did integration and unit testing alternately, even within builds (5.7.3.1). Like unit testing, integration and integration testing were more difficult for Ada than for FORTRAN. Issues of primary concern which arose were efficiency (both), interfaces (both), and strong typing, tasking, and exceptions (Ada). (See 4.7.4 and subsections, and 5.7.4 and subsections).

(11) How did all these processes differ for FORTRAN and Ada developments? What effect did these processes have on Ada products such as documentation and code?

The philosophies differed for each. The Ada team used data abstraction, information hiding, and the state machine concept; the FORTRAN team used structural decomposition and procedural abstraction (See 3.3). The design representations differed (object diagrams vs. structure charts) along with the methodologies (See 4.5 and 4.6). Management did not understand the GOOD notation at reviews (4.6.2), which they interpreted as structure charts. Coding from the design documents was especially easy with Ada, except for tasks (4.7.1.3 and 5.7.1.3). Question 10 includes the differences between Ada and FORTRAN for code reading, unit testing, integration and integration

testing. Psychologically, the tendency exists to trust the Ada compiler too much to discover faults. New intuitions must also be developed for discovering faults with Ada (See 4.7.3.4 and 5.7.3.4). FORTRAN was much more transparent to the developer than Ada. Reuse is important in this environment, and there was much reuse of design and code with FORTRAN; Ada had very little code reuse and no design reuse (See 4.9.2 and 5.9.2), since this was the first dynamics satellite simulator.

(12) How are all the activities and phases to be defined for Ada developments? How does this compare to the activities and phases in FORTRAN developments?

Much more overlap of activities occurred with the Ada project than with the FORTRAN project, and Ada phase boundaries were very fuzzy (See 4.6.3 and 5.6.3). Two extra phases were added to the Ada development where much overlap occurred (See 4.9.4 and 5.9.4). The percentage of time spent on the various activities was similar for both projects (See 4.9.3 and 5.9.3). One change recommended is to have CDR later (implicitly, a longer Design phase), or perhaps multiple reviews (See 4.6.3 and 5.6.3). Besides including answers for questions 7 through 10, when formulating new descriptions for Ada activities and phases, strategies need to be included for dealing with issues that do not arise in FORTRAN developments, such as recompilation (See 4.7.1.12 and 5.7.1.12).

II. Domain Conformance (Application domain, and developers' knowledge of it)

(1) How well did the Ada developers know the application domain? How did this compare to the application knowledge of the FORTRAN team?

The FORTRAN team had more experience with this type of application (dynamics simulators) than the Ada team (See 3.3).

(2) What kinds of development experience do the members of the Ada and FORTRAN teams have? How does this experience compare?

The Ada team had more overall development experience, and experience with more languages (See 3.3).

III. Effect (What happened)

(1) What effect did the FORTRAN biases in the specifications have on the Ada development process and product?

Since part of the mandate of the Ada team was to experiment with various design methodologies, it certainly was detrimental to have a high level FORTRAN design in the specifications. In addition, a design compatible with the Ada development methodologies was desired, especially if it was to be reused later on. The Composite Specification Model (CSM) was used to remove the design bias, as part of requirements analysis activity (See 3.3 and 4.4). Other benefits to the team included a better understanding of the requirements for the system.

(2) What are the effects of Ada on the Flight Dynamics development process, and the resulting product quality? How did Ada affect the following, and how does it compare to FORTRAN?

(a) the way design was done,

The design process and experiment with the three OOD methodologies are discussed in 4.5, 4.6, 5.5 and 5.6, and subsections. Questions I.8 and I.9 compare Ada and FORTRAN design issues. Some redesign was done of some pieces of the system during the implementation phase, once more experience had been gained with Ada (first project

effect). (See 4.7.1.4).

(b) the way implementation was done,

Ada and FORTRAN implementations were superficially similar, with builds, code reading, unit testing and integration (See 4.7). Except for tasks, it was easy to code from the design documents (See 4.7.1.3 and 5.7.1.3). Using library units or nesting is an implementation issue appearing only with Ada (See 4.7.1.5 and 5.7.1.5); nesting had many disadvantages. Many tasking issues also arose here. Better planning for tasks should have been done during design (See 4.7.1.6 and 5.7.1.6). Generics and separate compilations were easy to use and very useful (See 4.7.1.7 and 5.7.1.7). Though the interfaces were easier to design in Ada than they usually are in FORTRAN, many changes had to be made to them. The Ada design was new, and this was part of the problem. In addition, strong typing and the use of global types meant changes could have a large impact (See 4.7.1.8, 4.7.1.9, 4.7.1.10, 5.7.1.8, 5.7.1.9, 5.7.1.10). Meetings were more necessary for the Ada team than even for the FORTRAN team, in order to discuss new issues with Ada (e.g., recompilation), clarify misunderstandings (e.g., procedure function: see 4.7.1.11 and 4.7.1.12), and exchange new things learned (See 4.7.1.12 and 5.7.1.12). The Ada library structure was a lot more complex than FORTRAN (See 4.7.1.13 and 5.7.1.13). "Call–through" units, a unit with no procedural code that calls another unit, increases code size, causing several other problems, and should be avoided (See 4.7.1.14 and 5.7.1.14).

(c) the way testing was done,

Unit testing in both FORTRAN and Ada was done by the same individual who developed the code unit (See 4.7). Heavy nesting adversely affected unit testing and integration (See 4.7.1.5, 5.7.1.5, 4.7.3.1, 5.7.3.1, 4.7.3.2, 5.7.3.2). Integration was made more difficult by tasking (4.7.4.6 and 5.7.4.6), occasional misuse of exceptions (4.7.4.5 and 5.7.4.5), and nesting (4.7.4.4 and 5.7.4.4). Efficiency issues appear at this point for both FORTRAN and Ada developments (See 4.7.4.3 and 5.7.4.3).

(d) the products of each phase,

Design was positively affected by the rewritten specifications (See 4.4, 5.4, 4.6.1, 5.6.1). The design documentation for the most part, was quite helpful for representing the design and from which to develop code, though some problems existed. Task representation was a problem, and the design documentation needs some revision. Managers misunderstood the design documentation, although training can overcome that (see 4.6.2 and 5.6.2). Code was affected in many ways, both good and bad, by the various Ada features. With additional planning, primarily in design, many of these problems could be overcome (tasking – 4.7.1.6, 5.7.1.6, 4.7.4.6, 5.7.4.6; generics/separate compilations – 4.7.1.7, 5.7.1.7; strong typing – 4.7.1.10, 5.7.1.10; exceptions – 4.7.4.5, 5.7.4.5). Code is also affected by the balance of library units vs. nesting (4.7.1.5, 5.7.1.5, 4.7.4.4, 5.7.4.4), use of global types (4.7.1.9 and 5.7.1.9), use of "call–through" units (4.7.1.14 and 5.7.1.14), and use of non–portable features (4.7.1.15 and 5.7.1.15).

(e) the amount of effort spent in each phase, and activities during that phase,

The Ada project took a total of about 23,000 manhours; the FORTRAN project took about 15,000 manhours (See 4.2). Section 4.9.4 describes the effort in each phase and activities in each phase (See also 5.9.4).

(f) the amount of effort spent on each activity,

A different data form was used to collect activity data (rather than phase data), so total effort recorded by activity vs. by phase for each project is similar, but does not quite match. The percentage of time spent in each activity is similar for both the FORTRAN and Ada projects. This is true even if training (Ada) and Acceptance Testing (FORTRAN) are excluded. More time is spent in implementation in both projects

than in any other activity (See 4.9.3 and 5.9.3).

(g) the quality of the products:

(i) How many changes and failures were there? Were there fewer changes/failures with Ada?

About twice as many changes were made in the Ada system as in the FORTRAN system. A bit more than twice as many failures were found and corrected in the Ada system compared to the FORTRAN system. Sections 4.9.6 and 4.9.7 give distributions for the changes and failures in both systems (See also 5.9.6 and 5.9.7).

(ii) Why were the changes made?

The majority of changes in both systems (about 2/5) were to correct faults. Section 4.9.6 gives all the reasons for changes, and the distributions for the whole project and for each phase (See also 5.9.6). Reasons for interface changes are given in 4.7.1.8 and 5.7.1.8.

(iii) Where in the development process did the faults originate that eventually led to failures?

Most failures originate from coding errors. For the Ada project, errors in design and in making previous changes were also factors. (See Table 4.20 in 4.9.7, and also 5.9.7).

(iv) What type of failures occurred?

For both projects, data value or data structure problems and internal interface problems were the most frequent. Logic/control structure problems was an important reason also for problems in the Ada project. (See Table 4.21 in 4.9.7, and also 5.9.7). Isolating the reason for failures in tasks was particularly difficult (See 4.7.4.6 and 5.7.4.6).

(v) How hard (costly) were changes/failures to isolate and fix?

Effort to isolate changes was about the same for FORTRAN and Ada. Effort required to make the changes was more in FORTRAN (See Tables 4.17 and 4.18 in 4.9.6, and also 5.9.6). Effort to isolate failures was less for FORTRAN, however, effort required to correct failures was about the same for FORTRAN and Ada (See Tables 4.22 and 4.23 in 4.9.7, and also 5.9.7). The hardest type to fix were problems masked by exceptions during integration (4.7.4.5 and 5.7.4.5), and task problems, particularly when exceptions were also involved (4.7.4.6 and 5.7.4.6).

(3) How were the FORTRAN and Ada designs different? The same?

Many similarities exist between the designs, since they are solving the same problem. However, important differences also exist. Functions are distributed differently into various subsystems, the degree of coupling between the units is greater in the Ada system, the nature of the data flows and timing of subsystems is different, and the Truth Model is different since the Ada version does not have the real-time constraint the FORTRAN version had (See 4.6.1 and 5.6.1).

(4) How do we compare FORTRAN and Ada products? What measures can validly compare things such as size and productivity?

The FORTRAN and Ada products are compared here by size and effort (4.2, 4.9.1, 5.9.1, 4.9.3, 4.9.4), amount of reuse (4.9.2 and 5.9.2), and productivity (4.9.5 and 5.9.5). "Source lines of code" comparisons have limited usefulness between projects in different languages (5.2).

(5) What effect did the various Ada features have on the resulting system?

(a) generics

Generics were easy to implement, and reduced the amount of source code required (See 4.7.1.7 and 5.7.1.7).

(b) separate compilations for bodies and specifications

These are also easily implemented and beneficial (See 4.7.1.7 and 5.7.1.7). One benefit is that a library of Ada specifications can be made during design, before implementation begins, and these specifications can be stubs during testing for the units not yet implemented (See 5.7.1.13).

(c) library units vs. nesting

Many disadvantages were found to nesting, and many advantages to library units during coding, unit testing, and integration (See 4.7.1.5, 5.7.1.5, 4.7.3.1, 5.7.3.1, 4.7.3.2, 5.7.3.2, 4.7.4.4, 5.7.4.4).

(d) tasking

Concurrency is difficult in its own right, and caused many problems for the team in coding, unit testing, and integration (See 4.7.1.6, 5.7.1.6, 4.7.4.6, 5.7.4.6, 5.7.3.1).

(e) exceptions

Finding problems could be helped or hindered by use of exceptions, depending on how it was done (See 4.7.4.5 and 5.7.4.5). They were especially difficult to use in tasks, since there are some ways their behavior is different inside these units (See 5.7.3.1).

(f) strong typing

Strong typing had some benefit in finding faults early, but also led to problems. The global type package (4.7.1.9 and 5.7.1.9) did not work so well with a strongly typed language as it did with a weakly typed one. Global types are also a disadvantage due to the time recompilation takes with changes. Interface problems and type proliferation were also problems (4.7.1.8, 5.7.1.8, 4.7.1.10, 5.7.1.10). Unit testing was also a problem due to the need to create I/O procedures for all the types, and increased complexity of test drivers (4.7.3.3 and 5.7.3.3).

(6) What was expected to happen (with either the development process or the resulting system)? What did happen? Why the discrepancy between expectations and reality, if there is one?

At the top level, the prescriptive development process for Ada (3.2) and the size and effort estimates (4.2 and 5.2) show what was expected to happen. Chapters 4 and 5 describe what did happen. In particular, we note the amount of reuse (4.9.2), product size (4.9.1 and 5.9.1), effort by activity (4.9.3 and 5.9.3), effort by phase (4.9.4 and 5.9.4), and characteristics of each design (4.6.1 and 5.6.1). We note particularly as unexpected the disadvantages of nesting (4.7.1.5, 5.7.1.5, 4.7.4.4, 5.7.4.4), task proliferation (4.7.1.6 and 5.7.1.6), type proliferation (4.7.1.10 and 5.7.1.10), disadvantages of global types (4.7.1.9 and 5.7.1.9), the unexpected difficulties of unit testing (4.7.3, 5.7.3, and subsections), the unexpected difficulties of integration and integration testing (4.7.4, 5.7.4, and subsections).

(7) Is it feasible and cost effective to use Ada (in this kind of environment)?

At least some significant part of the extra time the Ada project took in every activity is due to the newness of Ada here (See 4.9.3 and 5.9.3). The FORTRAN system took 85% of the effort the Ada system took to develop (See 4.9.5 and 5.9.5), when we make adjustments for the different amounts of reuse in each system. Given that the FORTRAN development process is established, and the Ada development process is new, there is plenty of room for the Ada process to improve. Reuse for subsequent Ada projects is reported to be exceeding reuse on FORTRAN projects now.

(8) Switching from FORTRAN to Ada means losing the benefit of experience, institutional knowledge (which is no where written down, but necessary to operations), and reuse of designs and code. Do the benefits of using Ada compensate for these losses?

We cannot answer this yet, without the benefit of maintenance data, and studies of subsequent projects, which is beyond the scope here.

IV. Feedback (What should be done next time)

(1) What kind of training is needed in order to develop systems well with Ada?

Recommendations are given in 5.3. In addition to the training given to the development team, training managers so that they understand the notation at design reviews would greatly help communication (See 4.6.2 and 5.6.2). Code reading Ada code is also very useful for training (See 5.7.2).

(2) If the effect of the FORTRAN biases in the specifications is negative, how should the process be changed to avoid the FORTRAN bias? Would a bias toward Ada be a good thing?

Design issues began to surface in the Requirements Analysis phase. To eliminate the FORTRAN bias, rewriting the specifications with a methodology that would yield no bias, or an OOD bias, was required. CSM (Composite Specification Model) was used for this purpose (See 4.4 and 5.4).

(3) How should documentation problems be dealt with? What tailoring of object oriented methodologies is required for this environment? Which design method is appropriate for the specific application, and can it be scaled up to the problem size?

GOOD (General Object Oriented Design) is a methodology tailored to this environment (See 4.5, 4.6, and 5.6). This methodology works well for a scientific, non–real time project that primarily uses sequential code (See 5.5). Some changes were needed for representing tasking in designs (See 4.6.2, 5.6.2, 4.7.1.6, 5.7.1.6), but overall, it was easy to use the design documents for coding when the principles of OOD were understood (See 5.6.2, 4.7.1.3, 5.7.1.3). Also, the prologs that had descriptions given and no algorithm given were not as clear (See 4.7.1.11 and 5.7.1.11).

(4) How should the existing development process be modified to best change from FORTRAN to Ada?

(a) requirements analysis

Consider if detrimental bias exists in the incoming specifications. If so, use some method to rewrite the specifications to remove the bias (See 5.4).

(b) design

CDR should be later, or perhaps multiple design reviews. This would allow compilable PDL (See 4.6.3 and 5.6.3). A global analysis and overview of all tasks in the system and how they interact should be part of the design documentation, to prevent task proliferation (See 4.7.1.6 and 5.7.1.6). As few global types as possible should be planned (See 4.7.1.9 and 5.7.1.9). Type proliferation can be controlled by incorporating a data type analysis into design (See 4.7.1.10 and 5.7.1.10). The design needs to specify functions down to the procedural level and not just package level (See 5.7.1.12). Exceptions should also be planned as part of the abstractions created in design, and not added as an afterthought during implementation (See 4.7.4.5 and 5.7.4.5).

(c) implementation, code

Bottom–up implementation might be easier, since it correlates with the way Ada dependencies work. This could aid unit testing and integration later (See 5.7). Having a library of specifications ready when implementation starts would also make implementation easier (See 5.7.1.13). Since recompilation can be time consuming, it is important to plan for it (See 4.7.1.12 and 5.7.1.12). In addition, there are many coding practices that would promote reuse: separate general and application specific utilities

into different packages (4.7.1.1 and 5.7.1.1), use library units liberally and nesting sparingly (4.7.1.5 and 5.7.1.5), keep the global types package as small as possible, place types as "low" in the hierarchy as possible (4.7.1.9 and 5.7.1.9), limit code size through use of generics (4.7.1.7 and 5.7.1.7), do not use "call–through" units (4.7.1.14 and 5.7.1.14), and have a data type analysis (in design) (See 4.7.1.10 and 5.7.1.10).

(d) implementation, code reading

Code reading should have a different emphasis with Ada developments. Ada code is not automatically more readable; certain styles promote readability. Code reading is useful for teaching Ada as well as discovering faults (See 4.7.2 and 5.7.2).

(e) implementation, unit testing, integration and integration testing

Tasking, strong typing, exception handling and nesting are the Ada features that caused the most trouble for the team during unit testing and integration (See 4.7.3.1 and 5.7.3.1). Alternating integration and unit testing is best for testing with Ada (See 4.7.3.1 and 5.7.3.1).

(5) What unexpected problems have been encountered in development? What ways have we found to deal with them?

The major unexpected problems were (1) the disadvantages of nesting (4.7.1.5 and 5.7.1.5), (2) task proliferation (4.7.1.6 and 5.7.1.6), (3) the disadvantage of many global types (4.7.1.9 and 5.7.1.9), (4) type proliferation (4.7.1.10 and 5.7.1.10), (5) the problem of recompilation (time consuming – 4.7.1.12 and 5.7.1.12), and (6) the difficulty of unit testing and integration (4.7.3, 4.7.3.1, 5.7.3, 5.7.3.1).

CHAPTER 7

Future Research

Further analysis of the data contained here is planned. The results in chapters four and five can be turned into a succinct list of recommendations, and characteristics that the Ada life cycle should have. From this a model for Ada developments can be derived.

Open questions still exist on several fronts. One problem still to be solved is finding measures for comparing projects done in different languages. In particular, product measures such as SLOC, or even statements, are not equivalent between languages when considering effort or productivity. This is even more important in light of the need for increasing productivity as the demand for software continues to grow.

There is also a need for more case studies of Ada developments. We need to learn how other experiences are similar and different, and what factors affect the various process and product characteristics. We can only hypothesize these relationships, and then test them, if we have more data.

Another open question is a methodological one with lessons learned case studies. The lessons learned are closely related to the "story", that is, what happened during development. This makes presentation very difficult. The goal is to separate out the actual data from the conclusions. This makes it clearer for the reader which is which, and it also keeps the reader from being lost in the data while reading the lessons. On the other hand, the basis for the lessons needs to be easily found among the data. The data and lessons are closely intertwined sometimes, and this is also true for some of the different subjects discussed. Thus, it is hard not to be repetitive. These are the difficulties when presenting these kinds of results in written form.

There are more questions when presenting such data and conclusions in an automated system. This could be part of a system where the improvement paradigm[14] has been at least partially automated in a particular development environment. The basic problem above stems from the linearity of written text. That is not a problem here; hypertext can be used for the many cross references which are required to find information that is related. However, there still are representation problems on two levels. This is mentioned in section 3.5. The first level is the individual lesson, and its supporting data. What types of information should be there, and how should it be organized? The second problem is to choose how these lessons should be linked to each other. Of all the possible organizations mentioned in chapter 3, which should be used? Should the links be static or dynamic? What are the criteria for determining these things? How should such a database, particularly if it is more sophisticated, and has the capacity to make inferences, be related to other tools in a development environment? Progress in these areas will hopefully bring us another step closer to improving software productivity and quality.

Appendix: Data Collection Forms

Four of the forms used to collect information for the SEL database are shown here. These are the versions in effect when data was collected for the FORTRAN and Ada projects discussed here; they have been revised since then. Two of the forms collect information at the component level. For FORTRAN, a component is considered to be a subprogram or COMMON block. A component is considered to be a task, package, subprogram, body or specification for Ada. A package with three small procedures, for example, may be considered a component at the package level, if that is the compilable unit the programmer used[1]. Small subprograms might be grouped into one component in a FORTRAN system also. The definition of a component in this environment is important primarily for data collection[1].

Resource Summary Form

Effort data is reported weekly, by person (developer, management, clerical), on this form.

Component Origination Form

One of these is filled out by the developer creating the particular unit in question, when it goes under configuration control. This form is only done once.

Component Status Report

Effort data is related to activity at the component level, on this report. The first two characters in the component name identify the subsystem the component is a part of. If the first two characters are "$$", the activity is charged to the whole project, not a subsystem.

Change Report Form

Changes, including errors/failures, are reported on this form whenever a change is required, for units already under configuration control.

RESOURCE SUMMARY

PROJECT _____ DATE _____

NAME _____

WEEK OF:											
MANPOWER (HOURS)											% OF MGMT.
COMPUTER USAGE (NO. RUNS/HOURS CHARGED)											
OTHER CHARGES TO PROJECT											

500-3 (6/78)

COMPONENT ORIGINATION FORM

Component _____ Programmer _____

Subsystem _____ Date _____

Project _____

Location of source file

Library or directory _____ Member name _____

Relative difficulty of component

Please indicate (your judgement) by marking an X on the line below:

_____

Easy Medium Hard

Origin

If the component was modified or derived from a different project, please indicate the approximate amount of change and from where it was acquired; if it was coded new (from detailed design) indicate NEW.

_____ NEW

_____ Extensively modified (more than 25% of statements changed)

_____ Slightly modified

_____ Old (Unchanged)

If not new, where is it from ? _____

Type of Component

_____ 'INCLUDE' file (e.g., COMMON)

_____ JCL (or other control)

_____ ALC (assembler code)

_____ FORTRAN executable source

_____ Pascal source

_____ Ada source

_____ Namelists or parameter lists

_____ Display identification (GESS)

_____ Menu definition or help

_____ Reference data files

_____ BLOCK DATA file

_____ other (describe) _____

Purpose of Executable Component

For executable code, please identify the major purpose or purposes of this component. (Check all that apply).

_____ I/O processing

_____ Algorithmic/computational

_____ Data transfer

_____ Logic/decision

_____ Driver module

_____ Interface to operating system

GSFC 550-1 (3/85)

COMPONENT STATUS REPORT

PROJECT _____ DATE _____

PROGRAMMER _____

COMPONENT	DESIGN			CODE DEVELOPMENT			TEST			OTHER	
	CREATE	READ	FORMAL REVIEW	CODE	READ	FORMAL REVIEW	UNIT	INTEG	REVIEW	ACTIVITY	HRS
										TRAVEL	
										FORMS	
										MEETINGS	
										ACC TEST	
										TRAINING	
JCL											
OVERLAY											
USER GUIDE											
SYSTEM DESC											
										$$ANALYT	
										$$BLKTIM	
										$$CONSUL	
										$$DATGEN	
										$$DATSET	
										$$DEMO	
										$$IMPLAN	
										$$INTERF	
										$$KEYPCH	
										$$MANUAL	
										$$MEMO	
										$$MNTHLY	
										$$NOTEBK	
										$$PAPERW	
										$$PLANS	
										$$PRESNT	
										$$QUESTS	
										$$ROSW	
										$$RREQS	
										$$RSTDS	
										$$SCHEDL	
										$$SEMINR	
										$$SIM	
										$$STATUS	
										$$SYSTAP	
										$$SYSTST	
										$$TESTPL	
										$$TOOL	
										$$WEEKLY	
										$$XEROX	

CHANGE REPORT FORM

PROJECT NAME _____ CURRENT DATE _____

PROGRAMMER NAME _____ APPROVED BY _____

SECTION A – IDENTIFICATION

DESCRIBE THE CHANGE: (What, why, how) _____

EFFECT: What components (or documents) are changed? (Include version) _____

EFFORT: What additional components (or documents) were examined in determining what change was needed? _____

	(Month)	Day	Year
Need for change determined on			
Change completed (incorporated into system)			

	1 hr/less	1 hr/1 dy	1 dy/3 dys	>3 dys
Effort in person time to isolate the change (or error)				
Effort in person time to implement the change (or correction)				

SECTION B – ALL CHANGES

TYPE OF CHANGE (Check one)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain on back)

EFFECTS OF CHANGE

Y	N	
☐	☐	Was the change or correction to one and only one component?
☐	☐	Did you look at any other component?
☐	☐	Did you have to be aware of parameters passed explicitly or implicitly (e.g., common blocks) to or from the changed component?

SECTION C – FOR ERROR CORRECTIONS ONLY

SOURCE OF ERROR (Check one)

- ☐ Requirements
- ☐ Functional specifications
- ☐ Design
- ☐ Code
- ☐ Previous change

CLASS OF ERROR (Check most applicable)*

- ☐ Initialization
- ☐ Logic/control structure (e.g., flow of control incorrect)
- ☐ Interface (internal) (module to module communication)
- ☐ Interface (external) (module to external communication)
- ☐ Data (value or structure) (e.g., wrong variable used)
- ☐ Computational (e.g., error in math expression)

*If two are equally applicable, check the one higher on the list.

CHARACTERISTICS (Check Y or N for all)

Y	N	
☐	☐	Omission error (e.g., something was left out)
☐	☐	Commission error (e.g., something incorrect was included)
☐	☐	Error was created by transcription (clerical)

FOR LIBRARIANS USE ONLY

NUMBER _____

DATE _____

BY _____

CHECKED BY _____

	(Month)	Day	Year
ORIGIN DATE			

(Additional Comments on Reverse Side)

Glossary

AAS	Advanced Automation System
AI	Artificial Intelligence
ACS	Ada Compilation system
AFATDS	Advanced Field Artillery Tactical Data System
APSE	Ada Program Support Environment
CCB	Configuration Control Board
CDR	Critical Design Review
CMS	Configuration Management System
CSC	Computer Sciences Corporation
CSM	Composite Specification Model
DEC	Digital Equipment Corporation
EMS	Electronic Mail System
FAA	Federal Aviation Administration
GOOD	General Object Oriented Design
G/Q/M	Goal/Question/Metric Paradigm
GRO	Gamma Ray Observatory
GROSIM	GRO Simulator
NASA	National Aeronautics and Space Administration
OBC	On–Board Computer
OOD	Object Oriented Design
ORR	Operational Readiness Review
PAMELA	Process Abstraction Method for Embedded Large Applications
PDL	Program Design Language
PDR	Preliminary Design Review
SCIO	Simulator Control and I/O
SEL	Software Engineering Laboratory
SIMCON	Simulation Control
SLOC	Source Lines of Code
SRR	System Requirements Review
TAME	Tailoring A Measurement Environment
TM	Truth Model
UI	User Interface

References

1. William Agresti, *Ada Experiment Study - Team Lessons Learned,* Computer Sciences Corporation report, prepared for NASA/Goddard Space Flight Center, August, 1985.

2. William Agresti, *Guidelines for Applying the Composite Specification Model (CSM),* SEL–87–003, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, June, 1987.

3. W. Agresti, F. McGarry, D. Card, J. Page, V. Church, and R. Werking, *Manager's Handbook for Software Development,* SEL–84–001, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, April, 1984.

4. W. Agresti, E. Brinker, P. Lo, R. Murphy, E. Seidewitz, D. Shank, and M. Stark, *GRO Dynamics Simulator in Ada (GRODY) Detailed Design Notebook,* prepared for: NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, March, 1986.

5. W. W. Agresti, V. E. Church, D. N. Card, and P. L. Lo, "Designing with Ada for Satellite Simulation: A Case Study," *Proceedings of the First International Conference on Ada Applications for the NASA Space Station,* June, 1986.

6. J. Bailey and V. Basili, "A Meta–Model for Software Development Resource Expenditures," *Proceedings of the Fifth International Conference on Software Engineering,* pp. 107–116, IEEE Computer Society, 1981.

7. Victor R Basili, Elizabeth E Katz, Nora M Panlilio–Yap, Connie L Ramsey, and Shih Chang, "Characterization of an Ada Software Development," *Computer* **18**(9), pp. 53–65, IEEE, September, 1985.

8. V. R. Basili and R. W.. Selby, "Data Collection and Analysis in Software Research and Management," *Proceedings of the American Statistical Association,* 1984.

9. V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering* **SE–10**(6), pp. 728–738, November 1984.

10. V. R. Basili, "Measuring the Software Process and Product: Lessons Learned in the SEL," *Proceedings of the Tenth Annual Software Engineering Workshop,* NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, December 4, 1985. (Listed in the proceedings as "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying")

11. V. R. Basili and C. L. Ramsey, "Arrowsmith–P – A Prototype Expert System for Software Engineering Management," *Proceedings of the Expert Systems in Government Symposium,* IEEE Computer Society, October, 1985.

12. V. R. Basili, "Quantitative Evaluation of Software Engineering Methodology," *First Pan Pacific Computer Conference,* September, 1985. (also available as a technical report, TR–1519, Department of Computer Science, University of Maryland at College Park, July, 1985)

13. V. R. Basili, B. W. Boehm, J. A. Clapp, D. Gaumer, M. Holden, and J. K. Summers, *Use of Ada for FAA's Advanced Automation System (AAS),* The MITRE Corporation, McLean, Virginia, April, 1987. (under contract to the FAA)

14. V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement–Oriented Software Environments," *IEEE Transactions on Software Engineering* **SE–14**(6), pp. 758–773, June, 1988.

15. J. Baskette, "Life Cycle Analysis of an Ada Project," *IEEE Software* **4**(1), pp. 40–47, January, 1987.

16. Grady Booch, *Software Engineering with Ada (1st and 2nd editions)*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1983, 1987.

17. C. Brophy, W. Agresti, and V. Basili, "Lessons Learned in Use of Ada Oriented Design Methods," *Proceedings of the Joint Ada Conference*, Arlington, Virginia, March 16–19, 1987.

18. C. Brophy, S. Godfrey, W. Agresti, and V. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," *Proceedings of the Sixth National Ada Conference*, Arlington, Virginia, March 14–17, 1988.

19. G. W. Cherry, *Advanced Software Engineering with Ada -- Process Abstraction Method for Embedded Large Applications*, Language Automation Associates, Reston, Virginia, 1985.

20. G. W. Cherry, *PAMELA Designer's Handbook*, Thought** Tools, 1986.

21. J. D. Gannon, E. E. Katz, and V. R. Basili, "Metrics for Ada Packages: An Initial Study," *Communications of the ACM* **29**(7), pp. 616–623, July, 1986.

22. S. Godfrey and C. Brophy, *Assessing the Ada Design Process and Its Implications: A Case Study*, SEL–87–004, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, July, 1987.

23. S. Godfrey and C. Brophy, "Experiences in the Implementation of a Large Ada Project," *Proceedings of the Washington Ada Symposium*, Tysons Corner, Virginia, June 1988.

24. S. Godfrey and C. Brophy, *Implementation of a Production Ada Project: The GRODY Study*, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, In preparation.

25. P. Jalote, "Functional Refinment and Nested Objects for Object–Oriented Design," *IEEE Transactions on Software Engineering* **15**(3), pp. 264–270, March 1989.

26. P. Kane, N. Leuci, and D. Reifer, "A Cost Model for Estimating the Costs of Developing Software in the Ada Programming Language," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Vol. 2*, IEEE Computer Society Press, January, 1988.

27. Software Engineering Laboratory (ed.), *Collected Software Engineering Papers: Volume V*, SEL–87–009, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, November, 1987.

28. F. McGarry, J. Page, S. Eslinger, V. Church, and P. Merwarth, *Recommended Approach to Software Development*, SEL–81–205, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, April, 1983.

29. F. McGarry and R. Nelson, *An Experiment with Ada -- The GRO Dynamics Simulator Project Plan*, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, April, 1985.

30. F. McGarry and D. Card, "Studies and Experiments in the Software Engineering Lab (SEL)," *Proceedings of the Tenth Annual Software Engineering Workshop*, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, December 4, 1985. (Listed in the proceedings as "Recent SEL Studies")

31. F. E. McGarry and W. W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," *Proceedings of the 21st Annual Hawaii*

International Conference on System Sciences, Vol. 2, pp. 302–310, IEEE Computer Society Press, January, 1988.

32. R. Murphy and M. Stark, *Ada Training Evaluation and Recommendations from the Gamma Ray Observatory Ada Development Team*, SEL–85–002, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, October, 1985.

33. W. Myers, ed., "Large Ada Projects Show Productivity Gains," *IEEE Software* **5**(6), p. 89, November, 1988.

34. V. Rajlich, "Paradigms for Design and Implementation in Ada," *Communications of the ACM* **28**(7), pp. 718–727, July, 1985.

35. H. D. Rombach, "Quantitative Assessment of Maintenance: An Industrial Case Study," *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society, September 21–24, 1987.

36. H. D. Rombach and B. T. Ulery, "Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL," *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society, October 16–19, 1989. (Available from the University of Maryland Department of Computer Science as Technical Report number CS–TR–2252)

37. D. Roy and A. Jaworski, "NASA's Software Engineering with Ada," *Aerospace America*, pp. 8–10, February, 1989.

38. J. E. Sammet, "Why Ada Is Not Just Another Programming Language," *Communications of the ACM* **29**(8), pp. 722–732, August, 1986.

39. E. Seidewitz and M. Stark, *General Object–Oriented Software Development*, SEL–86–002, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, August, 1986.

40. E. Seidewitz, W. Agresti, and D. Ferry, et. al., *Ada Style Guide (draft)*, Ada Users Group, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, July 1986.

41. E. Seidewitz, "General Object–Oriented Software Development: Background and Experience," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Vol. 2*, pp. 262–270, IEEE Computer Society Press, January, 1988.

42. J. Seigle, L. Esker, and Y. Shi, *System Testing of a Production Ada Project: The GRODY Study*, SEL–88–001, NASA/Goddard Space Flight Center, Greenbelt, Maryland 20771, March, 1988.

43. J. D. Valett and F. E. McGarry, "A Summary of Software Management Experiences in the Software Engineering Laboratory," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Vol. 2*, pp. 293–301, IEEE Computer Society Press, January, 1988.

44. D. M. Weiss and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," *IEEE Transactions on Software Engineering* **SE–11**(2), pp. 157–168, February, 1985.

45. R. Wilson, ed., "Ada's Influence Spreads Through the Defense Community," *Computer Design*, pp. 91–92, July, 1987.